

Minos Garofalakis
Johannes Gehrke
Rajeev Rastogi *Editors*

Data Stream Management

Processing High-Speed Data Streams

Data-Centric Systems and Applications

Series editors

M.J. Carey
S. Ceri

Editorial Board

A. Ailamaki
S. Babu
P. Bernstein
J.C. Freytag
A. Halevy
J. Han
D. Kossmann
I. Manolescu
G. Weikum
K.-Y. Whang
J.X. Yu

More information about this series at
<http://www.springer.com/series/5258>

Minos Garofalakis • Johannes Gehrke •
Rajeev Rastogi
Editors

Data Stream Management

Processing High-Speed Data Streams



Springer

Editors

Minos Garofalakis
School of Electrical and
Computer Engineering
Technical University of Crete
Chania, Greece

Rajeev Rastogi
Amazon India
Bangalore, India

Johannes Gehrke
Microsoft Corporation
Redmond, WA, USA

ISSN 2197-9723
Data-Centric Systems and Applications
ISBN 978-3-540-28607-3
DOI 10.1007/978-3-540-28608-0

ISSN 2197-974X (electronic)
ISBN 978-3-540-28608-0 (eBook)

Library of Congress Control Number: 2016946344

Springer Heidelberg New York Dordrecht London
© Springer-Verlag Berlin Heidelberg 2016

The fourth chapter in part 4 is published with kind permission of © 2004 Association for Computing Machinery, Inc.. All rights reserved.

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Data Stream Management: A Brave New World	1
Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi	
Part I Foundations and Basic Stream Synopses	
Data-Stream Sampling: Basic Techniques and Results	13
Peter J. Haas	
Quantiles and Equi-depth Histograms over Streams	45
Michael B. Greenwald and Sanjeev Khanna	
Join Sizes, Frequency Moments, and Applications	87
Graham Cormode and Minos Garofalakis	
Top-k Frequent Item Maintenance over Streams	103
Moses Charikar	
Distinct-Values Estimation over Data Streams	121
Phillip B. Gibbons	
The Sliding-Window Computation Model and Results	149
Mayur Datar and Rajeev Motwani	
Part II Mining Data Streams	
Clustering Data Streams	169
Sudipto Guha and Nina Mishra	
Mining Decision Trees from Streams	189
Geoff Hulten and Pedro Domingos	
Frequent Itemset Mining over Data Streams	209
Gurmeet Singh Manku	

Temporal Dynamics of On-Line Information Streams	221
Jon Kleinberg	

Part III Advanced Topics

Sketch-Based Multi-Query Processing over Data Streams	241
Alin Dobro, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi	
Approximate Histogram and Wavelet Summaries of Streaming Data	263
S. Muthukrishnan and Martin Strauss	
Stable Distributions in Streaming Computations	283
Graham Cormode and Piotr Indyk	
Tracking Queries over Distributed Streams	301
Minos Garofalakis	

Part IV System Architectures and Languages

STREAM: The Stanford Data Stream Management System	317
Arvind Arasu, Brian Babcock, Shiva Nath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom	
The Aurora and Borealis Stream Processing Engines	337
Uğur Çetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Samuel Madden, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Mike Stonebraker, Nesime Tatbul, Ying Xing, and Stan Zdonik	
Extending Relational Query Languages for Data Streams	361
N. Laptev, B. Mozafari, H. Mousavi, H. Thakkar, H. Wang, K. Zeng, and Carlo Zaniolo	
Hancock: A Language for Analyzing Transactional Data Streams	387
Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith	
Sensor Network Integration with Streaming Database Systems	409
Daniel Abadi, Samuel Madden, and Wolfgang Lindner	

Part V Applications

Stream Processing Techniques for Network Management	431
Charles D. Cranor, Theodore Johnson, and Oliver Spatscheck	
High-Performance XML Message Brokering	451
Yanlei Diao and Michael J. Franklin	
Fast Methods for Statistical Arbitrage	473
Eleftherios Soulas and Dennis Shasha	

Contents	vii
Adaptive, Automatic Stream Mining	499
Spiros Papadimitriou, Anthony Brockwell, and Christos Faloutsos	
Conclusions and Looking Forward	529
Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi	

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 (24×7) 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

J. Gehrke

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA
e-mail: johannes@microsoft.com

R. Rastogi

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

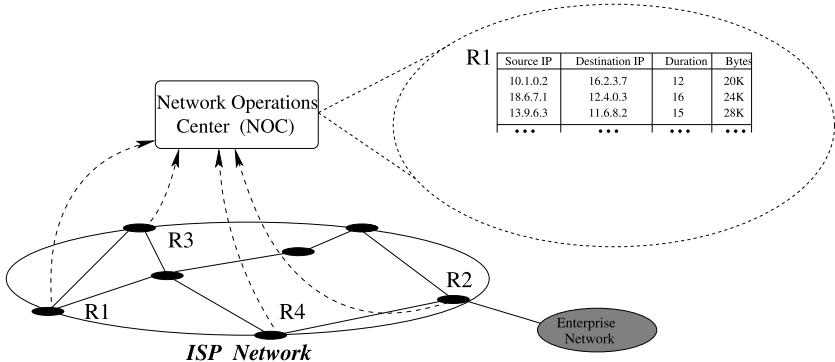


Fig. 1 ISP network monitoring data streams

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.

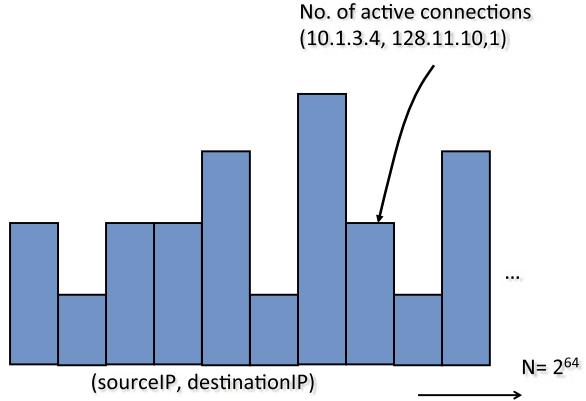
In the remainder of this section, we briefly outline some key data-stream management concepts and discuss basic stream-processing models.

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.¹ The dynamic vector A is rendered through

¹Note 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].

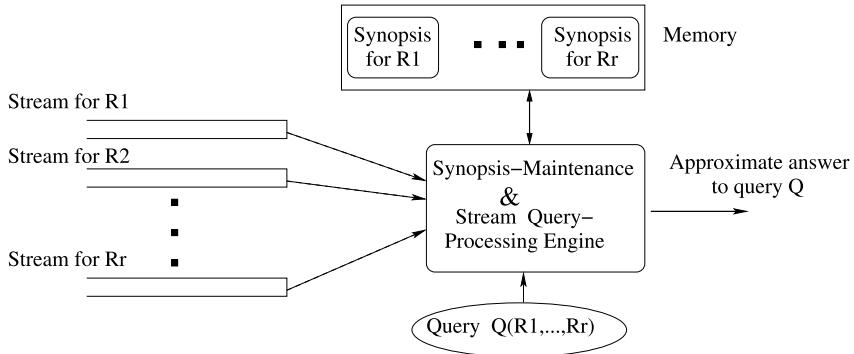


Fig. 3 General stream query processing architecture

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 ϵ relative/absolute error of the accurate answer) or *probabilistic* (e.g., estimate is within ϵ error of the accurate answer except for some small failure probability δ). The properties of such ϵ - or (ϵ, δ) -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

Part I

Foundations and Basic Stream Synopses

Data-Stream Sampling: Basic Techniques and Results

Peter J. Haas

1 Introduction

Perhaps the most basic synopsis of a data stream is a sample of elements from the stream. A key benefit of such a sample is its flexibility: the sample can serve as input to a wide variety of analytical procedures and can be reduced further to provide many additional data synopses. If, in particular, the sample is collected using random sampling techniques, then the sample can form a basis for statistical inference about the contents of the stream. This chapter surveys some basic sampling and inference techniques for data streams. We focus on general methods for materializing a sample; later chapters provide specialized sampling methods for specific analytic tasks.

To place the results of this chapter in context and to help orient readers having a limited background in statistics, we first give a brief overview of finite-population sampling and its relationship to database sampling. We then outline the specific data-stream sampling problems that are the subject of subsequent sections.

1.1 Finite-Population Sampling

Database sampling techniques have their roots in classical statistical methods for “finite-population sampling” (also called “survey sampling”). These latter methods are concerned with the problem of drawing inferences about a large finite *population* from a small random *sample* of population elements; see [1–5] for comprehensive

P.J. Haas (✉)
IBM Almaden Research Center, San Jose, CA, USA
e-mail: pshaas@us.ibm.com

discussions. The inferences usually take the form either of testing some hypothesis about the population—e.g., that a disproportionate number of smokers in the population suffer from emphysema—or estimating some parameters of the population—e.g., total income or average height. We focus primarily on the use of sampling for estimation of population parameters.

The simplest and most common sampling and estimation schemes require that the elements in a sample be “representative” of the elements in the population. The notion of *simple random sampling* (SRS) is one way of making this concept precise. To obtain an SRS of size k from a population of size n , a sample element is selected randomly and uniformly from among the n population elements, removed from the population, and added to the sample. This sampling step is repeated until k sample elements are obtained. The key property of an SRS scheme is that each of the $\binom{n}{k}$ possible subsets of k population elements is equally likely to be produced.

Other “representative” sampling schemes besides SRS are possible. An important example is *simple random sampling with replacement* (SRSWR).¹ The SRSWR scheme is almost identical to SRS, except that each sampled element is returned to the population prior to the next random selection; thus a given population element can appear multiple times in the sample. When the sample size is very small with respect to the population size, the SRS and SRSWR schemes are almost indistinguishable, since the probability of sampling a given population element more than once is negligible. The mathematical theory of SRSWR is a bit simpler than that of SRS, so the former scheme is sometimes used as an approximation to the latter when analyzing estimation algorithms based on SRS. Other representative sampling schemes besides SRS and SRSWR include the “stratified” and “Bernoulli” schemes discussed in Sect. 2. As will become clear in the sequel, certain non-representative sampling methods are also useful in the data-stream setting.

Of equal importance to sampling methods are techniques for estimating population parameters from sample data. We discuss this topic in Sect. 4, and content ourselves here with a simple example to illustrate some of the basic issues involved. Suppose we wish to estimate the total income θ of a population of size n based on an SRS of size k , where k is much smaller than n . For this simple example, a natural estimator is obtained by scaling up the total income s of the individuals in the sample, $\hat{\theta} = (n/k)s$, e.g., if the sample comprises 1 % of the population, then scale up the total income of the sample by a factor of 100. For more complicated population parameters, such as the number of distinct ZIP codes in a population of magazine subscribers, the scale-up formula may be much less obvious. In general, the choice of estimation method is tightly coupled to the method used to obtain the underlying sample.

Even for our simple example, it is important to realize that our estimate is *random*, since it depends on the particular sample obtained. For example, suppose (rather unrealistically) that our population consists of three individuals, say Smith, Abbas, and Raman, whose respective incomes are \$10,000, \$50,000, and

¹Sometimes, to help distinguish between the two schemes more clearly, SRS is called *simple random sampling without replacement*.

Table 1 Possible scenarios, along with probabilities, for a sampling and estimation exercise

Sample	Sample income	Est. Pop. income	Scenario probability
{Smith, Abbas}	\$60,000	\$90,000	1/3
{Smith, Raman}	\$1,010,000	\$1,515,000	1/3
{Abbas, Raman}	\$1,050,000	\$1,575,000	1/3

\$1,000,000. The total income for this population is \$1,060,000. If we take an SRS of size $k = 2$ —and hence estimate the income for the population as 1.5 times the income for the sampled individuals—then the outcome of our sampling and estimation exercise would follow one of the scenarios given in Table 1. Each of the scenarios is equally likely, and the expected value (also called the “mean value”) of our estimate is computed as

$$\begin{aligned} \text{expected value} &= (1/3) \cdot (90,000) + (1/3) \cdot (1,515,000) + (1/3) \cdot (1,575,000) \\ &= 1,060,000, \end{aligned}$$

which is equal to the true answer. In general, it is important to evaluate the accuracy (degree of systematic error) and precision (degree of variability) of a sampling and estimation scheme. The *bias*, i.e., expected error, is a common measure of accuracy, and, for estimators with low bias, the *standard error* is a common measure of precision. The bias of our income estimator is 0 and the standard error is computed as the square root of the *variance* (expected squared deviation from the mean) of our estimator:

$$\begin{aligned} \text{SE} &= [(1/3) \cdot (90,000 - 1,060,000)^2 + (1/3) \cdot (1,515,000 - 1,060,000)^2 \\ &\quad + (1/3) \cdot (1,575,000 - 1,060,000)^2]^{1/2} \approx 687,000. \end{aligned}$$

For more complicated population parameters and their estimators, there are often no simple formulas for gauging accuracy and precision. In these cases, one can sometimes resort to techniques based on *subsampling*, that is, taking one or more random samples from the initial population sample. Well known subsampling techniques for estimating bias and standard error include the “jackknife” and “bootstrap” methods; see [6]. In general, the accuracy and precision of a well designed sampling-based estimator should increase as the sample size increases. We discuss these issues further in Sect. 4.

1.2 Database Sampling

Although database sampling overlaps heavily with classical finite-population sampling, the former setting differs from the latter in a number of important respects.

- **Scarce versus ubiquitous data.** In the classical setting, samples are usually expensive to obtain and data is hard to come by, and so sample sizes tend to be small. In database sampling, the population size can be enormous (terabytes of data), and samples are relatively easy to collect, so that sample sizes can be relatively large [7, 8]. The emphasis in the database setting is on the sample as a flexible, lossy, compressed synopsis of the data that can be used to obtain quick approximate answers to user queries.
- **Different sampling schemes.** As a consequence of the complex storage formats and retrieval mechanisms that are characteristic of modern database systems, many sampling schemes that were unknown or of marginal interest in the classical setting are central to database sampling. For example, the classical literature pays relatively little attention to Bernoulli sampling schemes (described in Sect. 2.1 below), but such schemes are very important for database sampling because they can be easily parallelized across data partitions [9, 10]. As another example, tuples in a relational database are typically retrieved from disk in units of pages or extents. This fact strongly influences the choice of sampling and estimation schemes, and indeed has led to the introduction of several novel methods [11–13]. As a final example, estimates of the answer to an aggregation query involving select–project–join operations are often based on samples drawn individually from the input base relations [14, 15], a situation that does not arise in the classical setting.
- **No domain expertise.** In the classical setting, sampling and estimation are often carried out by an expert statistician who has prior knowledge about the population being sampled. As a result, the classical literature is rife with sampling schemes that explicitly incorporate auxiliary information about the population, as well as “model-based” schemes [4, Chap. 5] in which the population is assumed to be a sample from a hypothesized “super-population” distribution. In contrast, database systems typically must view the population (i.e., the database) as a black box, and so cannot exploit these specialized techniques.
- **Auxiliary synopses.** In contrast to a classical statistician, a database designer often has the opportunity to scan each population element as it enters the system, and therefore has the opportunity to maintain auxiliary data synopses, such as an index of “outlier” values or other data summaries, which can be used to increase the precision of sampling and estimation algorithms. If available, knowledge of the query workload can be used to guide synopsis creation; see [16–23] for examples of the use of workloads and synopses to increase precision.

Early papers on database sampling [24–29] focused on methods for obtaining samples from various kinds of data structures, as well as on the maintenance of sample views and the use of sampling to provide approximate query answers within specified time constraints. A number of authors subsequently investigated the use of sampling in query optimization, primarily in the context of estimating the size of select–join queries [22, 30–37]. Attention then shifted to the use of sampling to construct data synopses for providing quick approximate answers to decision-support queries [16–19, 21, 23]. The work in [15, 38] on online aggregation can be viewed

as a precursor to modern data-stream sampling techniques. Online-aggregation algorithms take, as input, streams of data generated by random scans of one or more (finite) relations, and produce continually-refined estimates of answers to aggregation queries over the relations, along with precision measures. The user aborts the query as soon as the running estimates are sufficiently precise; although the data stream is finite, query processing usually terminates long before the end of the stream is reached. Recent work on database sampling includes extensions of online aggregation methodology [39–42], application of bootstrapping ideas to facilitate approximate answering of very complex aggregation queries [43], and development of techniques for sampling-based discovery of correlations, functional dependencies, and other data relationships for purposes of query optimization and data integration [9, 44–46].

Collective experience has shown that sampling can be a very powerful tool, provided that it is applied judiciously. In general, sampling is well suited to very quickly identifying pervasive patterns and properties of the data when a rough approximation suffices; for example, industrial-strength sampling-enhanced query engines can speed up some common decision-support queries by orders of magnitude [10]. On the other hand, sampling is poorly suited for finding “needles in haystacks” or for producing highly precise estimates. The needle-in-haystack phenomenon appears in numerous guises. For example, precisely estimating the selectivity of a join that returns very few tuples is an extremely difficult task, since a random sample from the base relations will likely contain almost no elements of the join result [16, 31].² As another example, sampling can perform poorly when data values are highly skewed. For example, suppose we wish to estimate the average of the values in a data set that consists of 10^6 values equal to 1 and five values equal to 10^8 . The five outlier values are the needles in the haystack: if, as is likely, these values are not included in the sample, then the sampling-based estimate of the average value will be low by orders of magnitude. Even when the data is relatively well behaved, some population parameters are inherently hard to estimate from a sample. One notoriously difficult parameter is the number of distinct values in a population [47, 48]. Problems arise both when there is skew in the data-value frequencies and when there are many data values, each appearing a small number of times. In the former scenario, those values that appear few times in the database are the needles in the haystack; in the latter scenario, the sample is likely to contain no duplicate values, in which case accurate assessment of a scale-up factor is impossible. Other challenging population parameters include the minimum or maximum data value; see [49]. Researchers continue to develop new methods to deal with these problems, typically by exploiting auxiliary data synopses and workload information.

²Fortunately, for query optimization purposes it often suffices to know that a join result is “small” without knowing exactly how small.

1.3 Sampling from Data Streams

Data-stream sampling problems require the application of many ideas and techniques from traditional database sampling, but also need significant new innovations, especially to handle queries over infinite-length streams. Indeed, the unbounded nature of streaming data represents a major departure from the traditional setting. We give a brief overview of the various stream-sampling techniques considered in this chapter.

Our discussion centers around the problem of obtaining a sample from a *window*, i.e., a subinterval of the data stream, where the desired sample size is much smaller than the number of elements in the window. We draw an important distinction between a *stationary* window, whose endpoints are specified times or specified positions in the stream sequence, and a *sliding* window whose endpoints move forward as time progresses. Examples of the latter type of window include “the most recent n elements in the stream” and “elements that have arrived within the past hour.” Sampling from a finite stream is a special case of sampling from a stationary window in which the window boundaries correspond to the first and last stream elements. When dealing with a stationary window, many traditional tools and techniques for database sampling can be directly brought to bear. In general, sampling from a sliding window is a much harder problem than sampling from a stationary window: in the former case, elements must be removed from the sample as they expire, and maintaining a sample of adequate size can be difficult. We also consider “generalized” windows in which the stream consists of a sequence of transactions that insert and delete items into the window; a sliding window corresponds to the special case in which items are deleted in the same order that they are inserted.

Much attention has focused on SRS schemes because of the large body of existing theory and methods for inference from an SRS; we therefore discuss such schemes in detail. We also consider Bernoulli sampling schemes, as well as stratified schemes in which the window is divided into equal disjoint segments (the strata) and an SRS of fixed size is drawn from each stratum. As discussed in Sect. 2.3 below, stratified sampling can be advantageous when the data stream exhibits significant autocorrelation, so that elements close together in the stream tend to have similar values. The foregoing schemes fall into the category of *equal-probability sampling* because each window element is equally likely to be included in the sample. For some applications it may be desirable to bias a sample toward more recent elements. In the following sections, we discuss both equal-probability and biased sampling schemes.

2 Sampling from a Stationary Window

We consider a stationary window containing n elements e_1, e_2, \dots, e_n , enumerated in arrival order. If the endpoints of the window are defined in terms of time points t_1 and t_2 , then the number n of elements in the window is possibly random; this fact does not materially affect our discussion, provided that n is large enough so that

sampling from the window is worthwhile. We briefly discuss Bernoulli sampling schemes in which the size of the sample is random, but devote most of our attention to sampling techniques that produce a sample of a specified size.

2.1 Bernoulli Sampling

A *Bernoulli* sampling scheme with sampling rate $q \in (0, 1)$ includes each element in the sample with probability q and excludes the element with probability $1 - q$, independently of the other elements. This type of sampling is also called “binomial” sampling because the sample size is binomially distributed so that the probability that the sample contains exactly k elements is equal to $\binom{n}{k}q^k(1 - q)^{n-k}$. The expected size of the sample is nq . It follows from the central limit theorem for independent and identically distributed random variables [50, Sect. 27] that, for example, when n is reasonably large and q is not vanishingly small, the deviation from the expected size is within $\pm 100\epsilon$ % with probability close to 98 %, where $\epsilon = 2\sqrt{(1 - q)/nq}$. For example, if the window contains 10,000 elements and we draw a 1 % Bernoulli sample, then the true sample size will be between 80 and 120 with probability close to 98 %. Even though the size of a Bernoulli sample is random, Bernoulli sampling, like SRS and SRSWR, is a *uniform* sampling scheme, in that any two samples of the same size are equally likely to be produced.

Bernoulli sampling is appealingly easy to implement, given a pseudorandom number generator [51, Chap. 7]. A naive implementation generates for each element e_i a pseudorandom number U_i uniformly distributed on $[0, 1]$; element e_i is included in the sample if and only if $U_i \leq q$. A more efficient implementation uses the fact that the number of elements that are skipped between successive inclusions has a geometric distribution: if Δ_i is the number of elements skipped after e_i is included, then $\Pr\{\Delta_i = j\} = q(1 - q)^j$ for $j \geq 0$. To save CPU time, these random skips can be generated directly. Specifically, if U_i is a random number distributed uniformly on $[0, 1]$, then $\Delta_i = \lfloor \log U_i / \log(1 - q) \rfloor$ has the foregoing geometric distribution, where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x ; see [51, p. 465]. Figure 1 displays the pseudocode for the resulting algorithm, which is executed whenever a new element e_i arrives. Lines 1–4 represent an initialization step that is executed upon the arrival of the first element (i.e., when $m = 0$ and $i = 1$). Observe that the algorithm usually does almost nothing. The “expensive” calls to the pseudorandom number generator and the $\log()$ function occur only at element-inclusion times. As mentioned previously, another key advantage of the foregoing algorithm is that it is easily parallelizable over data partitions.

A generalization of the Bernoulli sampling scheme uses a different inclusion probability for each element, including element i in the sample with probability q_i . This scheme is known as *Poisson sampling*. One motivation for Poisson sampling might be a desire to bias the sample in favor of recently arrived elements. In general, Poisson sampling is harder to implement efficiently than Bernoulli sampling because generation of the random skips is nontrivial.

```

// q is the Bernoulli sampling rate
//  $e_i$  is the element that has just arrived ( $i \geq 1$ )
// m is the index of the next element to be included (static variable initialized to 0)
// B is the Bernoulli sample of stream elements (initialized to  $\emptyset$ )
// Δ is the size of the skip
// random() returns a uniform[0,1] pseudorandom number

1  if  $m = 0$  then                                //generate initial skip
2     $U \leftarrow \text{random}()$ 
3     $\Delta \leftarrow \lfloor \log U / \log(1 - q) \rfloor$ 
4     $m \leftarrow \Delta + 1$                          //compute index of first element to insert
5  if  $i = m$  then                                //insert element into sample and generate skip
6     $B \leftarrow B \cup \{e_i\}$ 
7     $U \leftarrow \text{random}()$ 
8     $\Delta \leftarrow \lfloor \log U / \log(1 - q) \rfloor$ 
9     $m \leftarrow m + \Delta + 1$                      //update index of next element to insert

```

Fig. 1 An algorithm for Bernoulli sampling

The main drawback of both Bernoulli and Poisson sampling is the uncontrollable variability of the sample size, which can become especially problematic when the desired sample size is small. In the remainder of this section, we focus on sampling schemes in which the final sample size is deterministic.

2.2 Reservoir Sampling

The reservoir sampling algorithm of Waterman [52, pp. 123–124] and McLeod and Bellhouse [53] produces an SRS of k elements from a window of length n , where k is specified a priori. The idea is to initialize a “reservoir” of k elements by inserting elements e_1, e_2, \dots, e_k . Then, for $i = k+1, k+2, \dots, n$, element e_i is inserted in the reservoir with a specified probability p_i and ignored with probability $1 - p_i$; an inserted element overwrites a “victim” that is chosen randomly and uniformly from the k elements currently in the reservoir. We denote by S_j the set of elements in the reservoir just after element e_j has been processed. By convention, we take $p_1 = p_2 = \dots = p_k = 1$. If we can choose the p_i ’s so that, for each j , the set S_j is an SRS from $U_j = \{e_1, e_2, \dots, e_j\}$, then clearly S_n will be the desired final sample. The probability that e_i is included in an SRS from U_i equals k/i , and so a plausible choice for the inclusion probabilities is given by $p_i = k/(i \vee k)$ for $1 \leq i \leq n$.³ The following theorem asserts that the resulting algorithm indeed produces an SRS.

Theorem 1 (McLeod and Bellhouse [53]) *In the reservoir sampling algorithm with $p_i = k/(i \vee k)$ for $1 \leq i \leq n$, the set S_j is a simple random sample of size $j \wedge k$ from $U_j = \{e_1, e_2, \dots, e_j\}$ for each $1 \leq j \leq n$.*

³Throughout, we denote by $x \vee y$ (resp., $x \wedge y$) the maximum (resp., minimum) of x and y .

Proof The proof is by induction on j . The assertion of the theorem is obvious for $1 \leq j \leq k$. Assume for induction that S_{j-1} is an SRS of size k from U_{j-1} , where $j \geq k+1$. Fix a subset $A \subset U_j$ containing k elements and first suppose that $e_j \notin A$. Then

$$\begin{aligned}\Pr\{S_j = A\} &= \Pr\{S_{j-1} = A \text{ and } e_j \text{ not inserted}\} \\ &= \binom{j-1}{k}^{-1} \frac{j-k}{j} = \binom{j}{k}^{-1},\end{aligned}$$

where the second equality follows from the induction hypothesis and the independence of the two given events. Now suppose that $e_j \in A$. For $e_r \in U_{j-1} - A$, let A_r be the set obtained from A by removing e_j and inserting e_r ; there are $j-k$ such sets. Then

$$\begin{aligned}\Pr\{S_j = A\} &= \sum_{e_r \in U_{j-1} - A} \Pr\{S_{j-1} = A_r, e_j \text{ inserted, and } e_r \text{ deleted}\} \\ &= \sum_{e_r \in U_{j-1} - A} \binom{j-1}{k}^{-1} \frac{k}{j} \frac{1}{k} = \binom{j-1}{k}^{-1} \frac{j-k}{j} = \binom{j}{k}^{-1}.\end{aligned}$$

Thus $\Pr\{S_j = A\} = 1/\binom{j}{k}$ for any subset $A \subset U_j$ of size k , and the desired result follows. \square

Efficient implementation of reservoir sampling is more complicated than that of Bernoulli sampling because of the more complicated probability distribution of the number of skips between successive inclusions. Specifically, denoting by Δ_i the number of skips before the next inclusion, given that element e_i has just been included, we have

$$f_i(m) \stackrel{\text{def}}{=} \Pr\{\Delta_i = m\} = \frac{k}{i-k} \frac{(i-k)^{\overline{m+1}}}{(i+1)^{\overline{m+1}}}$$

and

$$F_i(m) \stackrel{\text{def}}{=} \Pr\{\Delta_i \leq m\} = 1 - \frac{(i+1-k)^{\overline{m+1}}}{(i+1)^{\overline{m+1}}},$$

where $x^{\bar{n}}$ denotes the rising power $x(x+1)\cdots(x+n-1)$. Vitter [54] gives an efficient algorithm for generating samples from the above distribution. For small values of i , the fastest way to generate a skip is to use the method of *inversion*: if $F_i^{-1}(x) = \min\{m : F_i(m) \geq x\}$ and U is a random variable uniformly distributed on $[0, 1]$, then it is not hard to show that the random variable $X = F_i^{-1}(U)$ has the desired distribution function F_i , as does $X' = F_i^{-1}(1-U)$; see [51, Sect. 8.2.1]. For larger values of i , Vitter uses an *acceptance-rejection* method [51, Sect. 8.2.4]. For this method, there must exist a probability density function g_i from which it is easy

to generate sample values, along with a constant c_i —greater than 1 but as close to 1 as possible—such that $f_i(\lfloor x \rfloor) \leq c_i g_i(x)$ for all $x \geq 0$. If X is a random variable with density function g and U is a uniform random variable independent of X , then $\Pr\{\lfloor X \rfloor \leq x \mid U \leq f_i(\lfloor X \rfloor)/c_i g_i(X)\} = F_i(x)$. That is, if we generate pairs (X, U) until the relation $U \leq f_i(\lfloor X \rfloor)/c_i g_i(X)$ holds, then the final random variable X , after truncation to the nearest integer, has the desired distribution function F_i . It can be shown that, on average, c_i pairs (X, U) need to be generated to produce a sample from F_i . As a further refinement, we can reduce the number of expensive evaluations of the function f_i by finding a function h_i “close” to f_i such that h_i is inexpensive to evaluate and $h_i(x) \leq f_i(x)$ for $x \geq 0$. Then, to test whether $U \leq f_i(\lfloor X \rfloor)/c_i g_i(X)$, we first test (inexpensively) whether $U \leq h_i(\lfloor X \rfloor)/c_i g_i(X)$. Only in the rare event that this first test fails do we need to apply the expensive original test. This trick is sometimes called the “squeeze” method. Vitter shows that an appropriate choice for c_i is $c_i = (i+1)/(i-k+1)$, with corresponding choices

$$g_i(x) = \frac{k}{i+x} \left(\frac{i}{i+x} \right)^k \quad \text{and} \quad h_i(m) = \frac{k}{i+1} \left(\frac{i-k+1}{i+m-k+1} \right)^{k+1}.$$

Note that

$$G_i(x) = \int_0^x g_i(u) du = 1 - \left(\frac{i}{i+x} \right)^k,$$

so that, if V is a uniform random variable, then $G_i^{-1}(1-V) = i(V^{-1/k} - 1)$ has density function g_i . Thus it is indeed easy to generate sample values from g_i .

Figure 2 displays the pseudocode for the overall algorithm; see [54] for a performance analysis and some further optimizations.⁴ As with the algorithm in Fig. 1, the algorithm in Fig. 2 is executed whenever a new element e_i arrives.

Observe that the insertion probability $p_i = k/(i \vee k)$ decreases as i increases so that it becomes increasingly difficult to insert an element into the reservoir. On the other hand, the number of opportunities for an inserted element e_i to be subsequently displaced from the sample by an arriving element also decreases as i increases. These two opposing trends precisely balance each other at all times so that the probability of being in the final sample is the same for all of the elements in the window.

Note that the reservoir sampling algorithm does not require prior knowledge of n , the size of the window—the algorithm can be terminated after any arbitrary number of elements have arrived, and the contents of the reservoir are guaranteed to be an SRS of these elements. If the window size is known in advance, then a variation of reservoir sampling, called *sequential sampling*, can be used to obtain the desired SRS of size k more efficiently. Specifically, reservoir sampling has a time complexity of $O(k + k \log(n/k))$ whereas sequential sampling has a complexity of $O(k)$. The

⁴We do not recommend the optimization given in Eq. (6.1) of [54], however, because of a potential bad interaction with the pseudorandom number generator.

```

// k is the size of the reservoir and n is the number of elements in the window
// ei is the element that has just arrived ( $i \geq 1$ )
// m is the index of the next element  $\geq e_k$  to be included (static variable initialized to k)
// r is an array of length k containing the reservoir elements
// Δ is the size of the skip
// α is a parameter of the algorithm, typically equal to  $\approx 22k$ 
// random() returns a uniform[0,1] pseudorandom number

1  if  $i < k$  then                                //initially fill the reservoir
2     $r[i] \leftarrow e_i$ 
3  if  $i \geq k$  and  $i = m$ 
4    //insert  $e_i$  into reservoir
5    if  $i = k$                                      //no ejection needed
6       $r[k] \leftarrow e_i$ 
7    else                                         //eject a reservoir element
8       $U \leftarrow \text{random}()$ 
9       $I \leftarrow 1 + \lfloor kU \rfloor$            //I is uniform on {1, 2, ..., k}
10      $r[I] \leftarrow e_i$ 
11  //generate the skip Δ
12  if  $i \leq \alpha$  then                         //use inverse transformation
13     $U \leftarrow \text{random}()$ 
14    find the smallest integer  $\Delta \geq 0$  such that
15     $(i + 1 - k)^{\Delta+1}/(i + 1)^{\Delta+1} \leq U$    //evaluate  $F_i^{-1}(1 - U)$ 
16  else
17    repeat                                       //use acceptance-rejection + squeezing
18       $V \leftarrow \text{random}()$ 
19       $X \leftarrow i(V^{-1/k} - 1)$                  //generate sample from  $g_i$  via inversion
20       $U \leftarrow \text{random}()$ 
21      if  $U \leq h_i(\lfloor X \rfloor)/c_i g_i(X)$  then break
22    until  $U \leq f_i(\lfloor X \rfloor)/c_i g_i(X)$ 
23     $\Delta \leftarrow \lfloor X \rfloor$ 
24  //update index of next element to insert
25   $m \leftarrow i + \Delta + 1$ 

```

Fig. 2 Vitter's algorithm for reservoir sampling

sequential-sampling algorithm, due to Vitter [55], is similar in spirit to reservoir sampling, and is based on the observation that

$$\tilde{F}_{ij}(m) \stackrel{\text{def}}{=} \Pr\{\tilde{\Delta}_{ij} \leq m\} = 1 - \frac{(j-i)\underline{m+1}}{\underline{j}^{m+1}},$$

where $\tilde{\Delta}_{ij}$ is the number of skips before the next inclusion, given that element e_{n-j} has just been included in the sample and that the sample size just after the inclusion of e_{n-j} is $|S| = k - i$. Here x^n denotes the falling power $x(x-1) \cdots (x-n+1)$. The sequential-sampling algorithm initially sets $i \leftarrow k$ and $j \leftarrow n$; as above, i represents the number of sample elements that remain to be selected and j represents the number of window elements that remain to be processed. The algorithm then (i) generates $\tilde{\Delta}_{ij}$, (ii) skips the next $\tilde{\Delta}_{ij}$ arriving elements, (iii) includes the next

arriving element into the sample, and (iv) sets $i \leftarrow i - 1$ and $j \leftarrow j - \tilde{\Delta}_{ij} - 1$. Steps (i)–(iv) are repeated until $i = 0$.

At each execution of Step (i), the specific method used to generate $\tilde{\Delta}_{ij}$ depends upon the current values of i and j , as well as algorithmic parameters α and β . Specifically, if $i \geq \alpha j$, then the algorithm generates $\tilde{\Delta}_{ij}$ by inversion, similarly to lines 13–15 in Fig. 2. Otherwise, the algorithm generates $\tilde{\Delta}_{ij}$ using acceptance–rejection and squeezing, exactly as in lines 17–23 in Fig. 2, but using either $c_1 = j/(j-i+1)$,

$$g_1(x) = \begin{cases} \frac{i}{j}(1 - \frac{x}{j})^{i-1} & \text{if } 0 \leq x \leq j; \\ 0 & \text{otherwise,} \end{cases}$$

and

$$h_1(m) = \begin{cases} \frac{i}{j}(1 - \frac{m}{j-i+1})^{i-1} & \text{if } 0 \leq m \leq j-i; \\ 0 & \text{otherwise,} \end{cases}$$

or $c_2 = (i/(i-1))((j-1)/j)$,

$$g_2(x) = \frac{i-1}{j-1} \left(1 - \frac{i-1}{j-1}\right)^m,$$

and

$$h_2(m) = \begin{cases} \frac{i}{j}(1 - \frac{i-1}{j-m})^m & \text{if } 0 \leq m \leq j-i; \\ 0 & \text{otherwise.} \end{cases}$$

The algorithm uses (c_1, g_1, h_1) or (c_2, g_2, h_2) according to whether $i^2/j \leq \beta$ or $i^2/j > \beta$, respectively. The values of α and β are implementation dependent; Vitter found $\alpha = 0.07$ and $\beta = 50$ optimal for his experiments, but also noted that setting $\beta \approx 1$ minimizes the average number of random numbers generated by the algorithm. See [55] for further details and optimizations.⁵

2.3 Other Sampling Schemes

We briefly mention several other sampling schemes, some of which build upon or incorporate the reservoir algorithm of Sect. 2.2.

Stratified Sampling

As mentioned before, a stratified sampling scheme divides the window into disjoint intervals, or strata, and takes a sample of specified size from each stratum. The

⁵As with the reservoir sampling algorithm in [54], we do not recommend the optimization in Sect. 5.3 of [55].

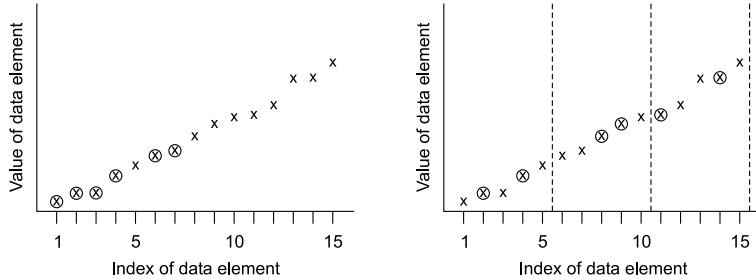


Fig. 3 (a) A realization of reservoir sampling (sample size = 6). (b) A realization of stratified sampling (sample size = 6)

simplest scheme specifies strata of approximately equal length and takes a fixed size random sample from each stratum using reservoir sampling; the random samples are of equal size.

When elements close together in the stream tend to have similar values, then the values within each stratum tend to be homogeneous so that a small sample from a stratum contains a large amount of information about all of the elements in the stratum. Figures 3(a) and 3(b) provide another way to view the potential benefit of stratified sampling. The window comprises 15 real-valued elements, and circled points correspond to sampled elements. Figure 3(a) depicts an unfortunate realization of an SRS: by sheer bad luck, the early, low-valued elements are disproportionately represented in the sample. This would lead, for example, to an underestimate of the average value of the elements in the window. Stratified sampling avoids this bad situation: a typical realization of a stratified sample (with three strata of length 5 each) might look as in Fig. 3(b). Observe that elements from all parts of the window are well represented. Such a sample would lead, e.g., to a better estimate of the average value.

Deterministic and Semi-Deterministic Schemes

Of course, the simplest scheme for producing a sample of size k inserts every m th element in the window into the sample, where $m = n/k$. There are two disadvantages to this approach. First, it is not possible to draw statistical inferences about the entire window from the sample because the necessary probabilistic context is not present. In addition, if the data in the window are periodic with a frequency that matches the sampling rate, then the sampled data will be unrepresentative of the window as a whole. For example, if there are strong weekly periodicities in the data and we sample the data every Monday, then we will have a distorted picture of the data values that appear throughout the week. One way to ameliorate the former problem is to use *systematic sampling* [1, Chap. 8]. To effect this scheme, generate a random number L between 1 and m . Then insert elements $e_L, e_{L+m}, e_{L+2m}, \dots, e_{n-m+L}$ into the sample. Statistical inference is now possible, but the periodicity issue still

remains—in the presence of periodicity, estimators based on systematic sampling can have large standard errors. On the other hand, if the data are not periodic but exhibit a strong trend, then systematic sampling can perform very well because, like stratified sampling, systematic sampling ensures that the sampled elements are spread relatively evenly throughout the window. Indeed, systematic sampling can be viewed as a type of stratified sampling where the i th stratum comprises elements $e_{(i-1)m+1}, e_{(i-1)m+2}, \dots, e_{im}$ and we sample one element from each stratum—the sampling mechanisms for the different strata are completely synchronized, however, rather than independent as in standard stratified sampling.

Biased Reservoir Sampling

Consider a generalized reservoir scheme in which the sequence of inclusion probabilities $\{p_i : 1 \leq i \leq n\}$ either is nondecreasing or does not decrease as quickly as the sequence $\{k/(i \vee k) : 1 \leq i \leq n\}$. This version of reservoir sampling favors inclusion of recently arrived elements over elements that arrived earlier in the stream.

As illustrated in Sect. 4.4 below, it can be useful to compute the marginal probability that a specified element e_i belongs to the final sample S . The probability that e_i is selected for insertion is, of course, equal to p_i . For $j > i \vee k$, the probability θ_{ij} that e_i is not displaced from the sample when element e_j arrives equals the probability that e_j is not selected for insertion plus the probability that e_j is selected but does not displace e_i . If $j \leq k$, then the processing of e_j cannot result in the removal of e_i from the reservoir. Thus

$$\theta_{ij} = (1 - p_j) + p_j \left(\frac{k-1}{k} \right) = \frac{k-p_j}{k}$$

if $j > k$, and $\theta_{ij} = 1$ otherwise. Because the random decisions made at the successive steps of the reservoir sampling scheme are mutually independent, it follows that the probability that e_i is included in S is the product of the foregoing probabilities:

$$\Pr\{e_i \in S\} = p_i \prod_{j=(i \vee k)+1}^n \frac{k-p_j}{k}. \quad (1)$$

Similar arguments lead to formulas for joint inclusion probabilities: setting $\alpha_{i,j} = \prod_{l=i}^j (k-p_l)/k$ and $\beta_{i,j} = \prod_{l=i}^j (k-2p_l)/k$, we have, for $i < j$,

$$\Pr\{e_i, e_j \in S\} = \begin{cases} p_i \alpha_{i+1, j-1} p_j ((k-1)/k) \beta_{j+1, n} & \text{if } k \leq i < j; \\ \alpha_{k+1, j-1} p_j ((k-1)/k) \beta_{j+1, n} & \text{if } i < k < j; \\ \beta_{k+1, n} & \text{if } i < j \leq k. \end{cases} \quad (2)$$

If, for example, we set $p_i \equiv p$ for some $p \in (0, 1)$. Then, from (1),

$$\Pr\{e_i \in S\} = p \left(\frac{k-p}{k} \right)^{n-(i \vee k)}.$$

Thus the probability that element e_i is in the final sample decreases geometrically as i decreases; the larger the value of p , the faster the rate of decrease.

Chao [56] has extended the basic reservoir sampling algorithm to handle arbitrary sampling probabilities. Specifically, just after the processing of element e_i , Chao's scheme ensures that the inclusion probabilities satisfy $\Pr\{e_j \in S\} \propto r_j$ for $1 \leq j \leq i$, where $\{r_j : j \geq 1\}$ is a prespecified sequence of positive numbers. The analysis of this scheme is rather complicated, and so we refer the reader to [56] for a complete discussion.

Biased Sampling by Halving

Another way to obtain a biased sample of size k is to divide the window into L strata of $m = n/L$ elements each, denoted $\Lambda_1, \Lambda_2, \dots, \Lambda_L$, and maintain a running sample S of size k as follows. The sample is initialized as an SRS of size k from Λ_1 ; (unbiased) reservoir sampling or sequential sampling may be used for this purpose. At the j th subsequent step, $k/2$ randomly-selected elements of S are overwritten by the elements of an SRS of size $k/2$ from Λ_{j+1} (so that half of the elements in S are purged). For an element $e_i \in \Lambda_j$, we have, after the procedure has terminated,

$$\Pr\{e_i \in S\} = \frac{k}{m} \left(\frac{1}{2}\right)^{L-(j \vee 2)+1}.$$

As with biased reservoir sampling, the halving scheme ensures that the probability that e_i is in the final samples falls geometrically as i decreases. Brönnimann et al. [57] describe a related scheme when each stream element is a d -vector of 0–1 data that represents, e.g., the presence or absence in a transaction of each of d items. In this setting, the goal of each halving step is to create a subsample in which the relative occurrence frequencies of the items are as close as possible to the corresponding frequencies over all of the transactions in the original sample. The scheme uses a deterministic halving method called “epsilon approximation” to achieve this goal. The relative item frequencies in subsamples produced by this latter method tend to be closer to the relative frequencies in the original sample than are those in subsamples obtained by SRS.

3 Sampling from a Sliding Window

We now restrict attention to infinite data streams and consider methods for sampling from a sliding window that contains the most recent data elements. As mentioned previously, this task is substantially harder than sampling from a stationary window. The difficulty arises because elements must be removed from the sample as they expire so that maintaining a sample of a specified size is nontrivial. Following [58], we distinguish between *sequence-based* windows and *timestamp-based* windows. A sequence-based window of length n contains the n most recent elements,

whereas a timestamp-based window of length t contains all elements that arrived within the past t time units. Because a sliding window inherently favors recently arrived elements, we focus on techniques for equal-probability sampling from within the window itself. For completeness, we also provide a brief discussion of *generalized* windows in which elements need not leave the window in arrival order.

3.1 Sequence-Based Windows

We consider windows $\{W_j : j \geq 1\}$, each of length n , where $W_j = \{e_j, e_{j+1}, \dots, e_{j+n-1}\}$. A number of algorithms have been proposed for producing, for each window W_j , an SRS S_j of k elements from W_j . The major difference between the algorithms lies in the tradeoff between the amount of memory required and the degree of dependence between the successive S_j 's.

Complete Resampling

At one end of the spectrum, a “complete resampling” algorithm takes an independent sample from each W_j . To do this, the set of elements in the current window is buffered in memory and updated incrementally, i.e., W_{j+1} is obtained from W_j by deleting e_j and inserting e_{j+n} . Reservoir sampling (or, more efficiently, sequential sampling) can then be used to extract S_j from W_j . The S_j 's produced by this algorithm have the desirable property of being mutually independent. This algorithm is impractical, however, because it has memory and CPU requirements of $O(n)$, and n is assumed to be very large.

A Passive Algorithm

At the other end of the spectrum, the “passive” algorithm described in [58] obtains an SRS of size k from the first n elements using reservoir sampling. Thereafter, the sample is updated only when the arrival of an element coincides with the expiration of an element in the sample, in which case the expired element is removed and the new element is inserted. An argument similar to the proof of Theorem 1 shows that each S_j is a SRS from W_j . Moreover, the memory requirement is $O(k)$, the same as for the stationary-window algorithms. In contrast to complete resampling, however, the passive algorithm produces S_j 's that are highly correlated. For example, S_j and S_{j+1} are identical or almost identical for each j . Indeed, if the data elements are periodic with period n , then every S_j is identical to S_1 ; this assertion follows from the fact that if element e_i is in the sample, then so is e_{i+jn} for $j \geq 1$. Thus if S_1 is not representative, e.g., the sampled elements are clustered within W_1 as in Fig. 3(a), then each subsequent sample will suffer from the same defect.

Subsampling from a Bernoulli Sample

Babcock et al. [58] provide two algorithms intermediate to those discussed above. The first algorithm inserts elements into a set B using a Bernoulli sampling scheme; elements are removed from B when, and only when, they expire. The algorithm tries to ensure that the size of B exceeds k at all times by using an inflated Bernoulli sampling rate of $q = (2ck \log n)/n$, where c is a fixed constant. Each final sample S_j is then obtained as a simple random subsample of size k from B . An argument using Chernoff bounds (see, e.g., [59]) shows that the size of B lies between k and $4ck \log n$ with a probability that exceeds $1 - O(n^{-c})$. The S_j 's are less dependent than in the passive algorithm, but the expected memory requirement is $O(k \log n)$. Also observe that if B_j is the size of B after j elements have been processed and if $\gamma(i)$ denotes the index of the i th step at which the sample size either increases or decreases by 1, then $\Pr\{B_{\gamma(i)+1} = B_{\gamma(i)} + 1\} = \Pr\{B_{\gamma(i)+1} = B_{\gamma(i)} - 1\} = 1/2$. That is, the process $\{B_{\gamma(i)} : i \geq 0\}$ behaves like a symmetric random walk. It follows that, with probability 1, the size of the Bernoulli sample will fall below k infinitely often, which can be problematic if sampling is performed over a very long period of time.

Chain Sampling

The second algorithm, called *chain sampling*, retains the improved independence properties of the S_j 's relative to the passive algorithm, but reduces the expected memory requirement to $O(k)$. The basic algorithm maintains a sample of size 1, and a sample of size k is obtained by running k independent chain-samplers in parallel. Observe that the overall sample is therefore a simple random sample with replacement—we discuss this issue after we describe the algorithm.

To maintain a sample S of size 1, the algorithm initially inserts each newly arrived element e_i into the sample (i.e., sets the sample equal to $S = \{e_i\}$) with probability $1/i$ for $1 \leq i \leq n$. Thus the algorithm behaves initially as a reservoir sampler so that, after the n th element has been observed, S is an SRS of size 1 from $\{e_1, e_2, \dots, e_n\}$. Subsequently, whenever element e_i arrives and, just prior to arrival, the sample is $S = \{e_j\}$ with $i = j + n$ (so that the sample element e_j expires), an element randomly and uniformly selected from among $e_{j+1}, e_{j+2}, \dots, e_{j+n}$ becomes the new sample element. Observe that the algorithm does not need to store all of the elements in the window in order to replace expiring sample elements—it suffices to store a “chain” of elements associated with the sample, where the first element of the chain is the sample itself; see Fig. 4. In more detail, whenever an element e_i is added to the chain, the algorithm randomly selects the index K of the element e_K that will replace e_i upon expiration. Index K is uniformly distributed on $i + 1, i + 2, \dots, i + n$, the indexes of the elements that will be in the window just after e_i expires. When element e_K arrives, the algorithm stores e_K in memory and randomly selects the index M of the element that will replace e_K upon expiration.

To further reduce memory requirements and increase the degree of independence between successive samples, the foregoing chaining method is enhanced with a

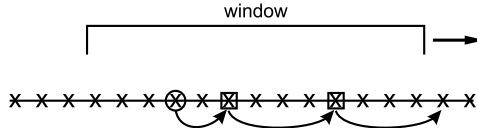


Fig. 4 Chain sampling (sample size = 1). Arrows point to the elements of the current chain, the *circled element* represents the current sample, and *elements within squares* represent those elements of the chain currently stored in memory

reservoir sampling mechanism. Specifically, suppose that element e_i arrives and, just prior to arrival, the sample is $S = \{e_j\}$ with $i < j + n$ (so that the sample element e_j does not expire). Then, with probability $1/n$, element e_i becomes the sample element; the previous sample element e_j and its associated chain are discarded, and the algorithm starts to build a new chain for the new sample element. With probability $1 - (1/n)$, element e_j remains as the sample element and its associated chain is not discarded. To see that this procedure is correct when $i < j + n$, observe that just prior to the processing of e_i , we can view S as a reservoir sample of size 1 from the “stream” of $n - 1$ elements given by $e_{i-n+1}, e_{i-n+2}, \dots, e_{i-1}$. Thus, adding e_i to the sample with probability $1/n$ amounts to executing a step of the usual reservoir algorithm, so that, after processing e_i , the set S remains an SRS of size 1 from the updated window $W_{i-n+1} = \{e_{i-n+1}, e_{i-n+2}, \dots, e_i\}$. Because the SRS property of S is preserved at each arrival epoch whether or not the current sample expires, a straightforward induction argument formally establishes that S is an SRS from the current window at all times.

Figure 5 displays the pseudocode for the foregoing algorithm; the code is executed whenever a new element e_i arrives. In the figure, the variable L denotes a linked list of chained elements of the form (e, l) , where e is an element and l is the element’s index in the stream; the list does not contain the current sample element, which is stored separately in S . Elements appear from head to tail in order of arrival, with the most recently arrived element at the tail of the list. The functions *add*, *pop*, and *purge* add a new element to the tail of the list, remove (and return the value of) the element at the head of the list, and remove all elements from the list, respectively.

We now analyze the memory requirements of the algorithm by studying the maximum amount of memory consumed during the evolution of a single chain.⁶ Denote by M the total number of elements inserted into memory during the evolution of the chain, including the initial sample. Thus $M \geq 1$ and M is an upper bound on the maximum memory actually consumed because it ignores decreases in memory consumption due to expiration of elements in the chain. Denote by X the distance from the initial sample to the next element in the chain, and recall that X is uniformly distributed on $\{1, 2, \dots, n\}$. Observe that $M \geq 2$ if and only if $X < n$ and, after the

⁶See [58] for an alternative analysis. Whenever an arriving element e_i is added to the chain and then immediately becomes the new sample element, we count this element as the first element of a new chain.

```

// n is the number of elements in the window
// ei is the element that has just arrived ( $i \geq 1$ )
// L is a linked list (static) of chained elements (excluding sample) of the form (e, l)
// S is the sample (static, contains exactly one element)
// J is the index of the element in the sample (static, initialized to 0)
// K is the index of the next element to be added to the chain (static, initialized to 0)
// random() returns a uniform[0,1] pseudorandom number

1   if i = K                                //add ei to chain
2     add(ei, i, L)                      //insert (ei, i) at tail of list
3     V ← random()
4     K ← i + ⌊nV⌋ + 1                     //K is uniform on i + 1, ..., i + n
5   if i = J + n                            //current sample element is expiring
6     (e, l) ← pop(L)                      //remove element at head of list...
7     S ← {e}
8     J ← l                                //... to become the new sample element
9   else                                     //sample element is not expiring
10    U ← random()
11    if U ≤ 1/(i ∧ n) then                //insert ei into sample
12      S ← {ei}
13      J ← i
14    purge(L)                             //start new chain
15    V ← random()
16    K ← i + ⌊nV⌋ + 1

```

Fig. 5 Chain-sampling algorithm (sample size = 1)

initial sample, none of the next X arriving elements become the new sample element. Thus $\Pr\{M \geq 2 \mid M \geq 1, X = j\} \leq (1 - n^{-1})^j$ for $1 \leq j \leq n$. Unconditioning on X , we have

$$\Pr\{M \geq 2 \mid M \geq 1\} \leq \sum_{j=1}^n \frac{1}{n} \left(1 - \frac{1}{n}\right)^j = 1 - \left(1 - \frac{1}{n}\right)^{n+1} \stackrel{\text{def}}{=} \beta.$$

The same argument also shows that $\Pr\{M \geq j + 1 \mid M \geq j\} \leq \beta$ for $j \geq 2$, so that $\Pr\{M \geq j\} \leq \beta^{j-1}$ for $j \geq 1$. An upper bound on the expected memory consumption is therefore given by

$$E[M] = \sum_{j=1}^{\infty} \Pr\{M \geq j\} \leq \frac{1}{1 - \beta} \approx e.$$

Moreover, for $j = \alpha \ln n$ with α a fixed positive constant, we have

$$\Pr\{M \geq j + 1\} = e^{j \ln \beta} = n^{-c},$$

where $c = -\alpha \ln \beta \approx -\alpha \ln(1 - e^{-1})$. Thus the expected memory consumption for k independent samplers is $O(k)$ and, with probability $1 - O(n^{-c})$, the memory consumption does not exceed $O(k \log n)$.

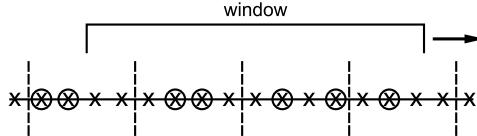


Fig. 6 Stratified sampling for a sliding window ($n = 12, m = 4, k = 2$). The *circled elements* lying within the window represent the members of the current sample, and *circled elements* lying to the left of the window represent former members of the sample that have expired

As mentioned previously, chain sampling produces an SRSWR rather than an SRS. One way of dealing with this issue is increase the size of the initial SRSWR sample S' to $|S'| = k + \alpha$, where α is large enough so that, after removal of duplicates, the size of the final SRS S will equal or exceed k with high probability. Subsampling can be then be used, if desired, to ensure that the final sample size $|S|$ equals k exactly. Using results on “occupancy distributions” [60, p. 102] it can be shown that

$$\Pr\{|S| < k\} = \sum_{j=1}^{k-1} \sum_{i=0}^j (-1)^i \binom{n}{n-j} \binom{j}{i} \left(1 - \frac{n-j+i}{n}\right)^{k+\alpha}, \quad (3)$$

and a value of α that makes the right side sufficiently small can be determined numerically, at least in principle. Assuming that $k < n/2$, a conservative but simpler approach ensures that $\Pr\{|S| < k\} < n^{-c}$ for a specified constant $c \geq 1$ by setting $\alpha = \alpha_1 \wedge \alpha_2$, where

$$\alpha_1 = \frac{(c-1) \ln n + (k+1) \ln k - (k-1) \ln 2}{\ln(n/k)}$$

and

$$\alpha_2 = c \ln n + (2ck \ln n + c^2 \ln^2 n)^{1/2}.$$

This assertion follows from a couple of simple bounding arguments.⁷

Stratified Sampling

The stratified sampling scheme for a stationary window can be adapted to obtain a stratified sample from a sliding window. The simplest scheme divides the stream into strata of length m , where m divides the window length n ; see Fig. 6. Reservoir sampling is used to obtain a SRS of size $k < m$ from each stratum. Sampled elements expire in the usual manner. The current window always contains between l and $l+1$ strata, where $l = n/m$, and all but perhaps the first and last strata are of equal length,

⁷We derive α_1 by directly bounding each term in (3). We derive α_2 by stochastically bounding $|S|$ from below by the number of successes in a sequence of $k + \alpha$ Bernoulli trials with success probability $(n-k)/n$ and then using a Chernoff bound.

namely m . The sample size fluctuates, but always lies between $k(l - 1)$ and kl . This sampling technique therefore not only retains the advantages of the stationary stratified sampling scheme but also, unlike the other sliding-window algorithms, ensures that the sample size always exceeds a specified threshold.

3.2 Timestamp-Based Windows

Relatively little is currently known about sampling from timestamp-based windows. The methods for sequence-based windows do not apply because the number of elements in the window changes over time. Babcock et al. [58] propose an algorithm called *priority sampling*. As with chain sampling, the basic algorithm maintains an SRS of size 1, and an SRSWR of size k is obtained by running k priority-samplers in parallel.

The basic algorithm for a sample size of 1 assigns to each arriving element a random priority uniformly distributed between 0 and 1. The current sample is then taken as the element in the current window having the highest priority; since each element in the window is equally likely to have the highest priority, the sample is clearly an SRS. The only elements that need to be stored in memory are those elements in the window for which there is no element with both a higher timestamp and a higher priority because only these elements can ever become the sample element. In one simple implementation, the stored elements (including the sample) are maintained as a linked list, in order of decreasing priority (and, automatically, of increasing timestamp). Each arriving element e_i is inserted into the appropriate place in the list, and all list elements having a priority smaller than that of e_i are purged, leaving e_i as the last element in the list. Elements are removed from the head of the list as they expire.

To determine the memory consumption M of the algorithm at a fixed but arbitrary time point, suppose that the window contains n elements $e_{m+1}, e_{m+2}, \dots, e_{m+n}$ for some $m \geq 0$. Denote by \mathcal{P}_i the priority of e_{m+i} , and set $\Phi_i = 1$ if e_{m+i} is currently stored in memory and $\Phi_i = 0$ otherwise. Ignore zero-probability events in which there are ties among the priorities and observe for each i that $\Phi_i = 1$ if and only if $\mathcal{P}_i > \mathcal{P}_j$ for $j = i + 1, i + 2, \dots, n$. Because priorities are assigned randomly and uniformly, each of the $n - i + 1$ elements $e_{m+i}, e_{m+i+1}, \dots, e_{m+n}$ is equally likely to be the one with the highest priority, and hence $E[\Phi_i] = \Pr\{\Phi_i = 1\} = 1/(n - i + 1)$. It follows that the expected number of elements stored in memory is

$$E[M] = E\left[\sum_{i=1}^n \Phi_i\right] = \sum_{i=1}^n E[\Phi_i] = H(n) = O(\ln n),$$

where $H(n)$ is the n th harmonic number. We can also obtain a probabilistic bound on M as follows. Denote by X_i the number of the i most recent arrivals in the window that have been inserted into the linked list: $X_i = \sum_{j=n-i+1}^n \Phi_j$. Observe

that if $X_i = m$ for some $m \geq 0$, then either $X_{i+1} = m$ or $X_{i+1} = m + 1$. Moreover, it follows from our previous analysis that $\Pr\{X_1 = 1\} = 1$ and

$$\begin{aligned} & \Pr\{X_{i+1} = m_i + 1 \mid X_i = m_i, X_{i-1} = m_{i-1}, \dots, X_1 = m_1\} \\ &= \Pr\{\Phi_{n-i} = 1\} \\ &= 1/(i+1) \end{aligned}$$

for all $1 \leq i < n$ and m_1, m_2, \dots, m_i such that $m_1 = 1$ and $m_{j+1} - m_j \in \{0, 1\}$ for $1 \leq j < i$. Thus $M = X_n$ is distributed as the number of successes in a sequence of n independent Poisson trials with success probability for the i th trial equal to $1/i$. Application of a simple Chernoff bound together with the fact that $\ln n < H(n) < 2 \ln n$ for $n \geq 3$ shows that $\Pr\{M > 2(1+c)\ln n\} < n^{-c^2/3}$ for $c \geq 0$ and $n \geq 3$. Thus, for the overall sampling algorithm the expected memory consumption is $O(k \log n)$ and, with high probability, memory consumption does not exceed $O(k \log n)$.

3.3 Generalized Windows

In the case of both sequence-based and timestamp-based sliding windows, elements leave the window in same order that they arrive. In this section, we briefly consider a generalized setting in which elements can be deleted from a window W in arbitrary order. More precisely, we consider a set $\mathcal{T} = \{t_1, t_2, \dots\}$ of unique, distinguishable *items*, together with an infinite sequence of transactions $\gamma = (\gamma_1, \gamma_2, \dots)$. Each transaction γ_i is either of the form $+t_k$, which corresponds to the insertion of item t_k into W , or of the form $-t_k$, which corresponds to the deletion of item t_k from W . We restrict attention to sequences such that, at any time point, an item appears at most once in the window, so that the window is a true set and not a multiset. To avoid trivialities, we also require that $\gamma_n = -t_k$ only if item t_k is in the window just prior to the processing of the n th transaction. Finally, we assume throughout that the rate of insertions approximately equals the rate of deletions, so that the number of elements in the window remains roughly constant over time.

The authors in [61] provide a “random pairing” (RP) algorithm for maintaining a bounded uniform sample of W . The RP algorithm generalizes the reservoir sampling algorithm of Sect. 2.2 to handle deletions, and reduces to the passive algorithm of Sect. 3.1 when the number of elements in the window is constant over time and items are deleted in insertion order (so that W is a sequence-based sliding window).

In the RP scheme, every deletion from the window is eventually “compensated” by a subsequent insertion. At any given time, there are 0 or more “uncompensated” deletions. The RP algorithm maintains a counter c_b that records the number of “bad” uncompensated deletions in which the deleted item was also in the sample so that the sample size was decremented by 1. The RP algorithm also maintains a counter c_g that records the number of “good” uncompensated deletions in which the deleted item was not in the sample so that the sample size was not affected. Clearly, $d = c_b + c_g$ is the total number of uncompensated deletions.

```

//  $c_b$  is the # of uncompensated deletions that have been in the sample
//  $c_g$  is the # of uncompensated deletions that have not been in the sample
//  $\gamma_i$  is the transaction that has just arrived ( $i \geq 1$ )
//  $M$  is the upper bound on sample size
//  $W$  and  $S$  are the window and sample size, respectively
//  $random()$  returns a uniform[0,1] pseudorandom number

1  if  $\gamma_i = +t$  then           //an insertion
2    if  $c_b + c_g = 0$           //execute reservoir-sampling step
3      if  $|S| < M$ 
4        insert  $t$  into  $S$ 
5      else if  $random() < M/(|W| + 1)$ 
6        overwrite a randomly selected element of  $S$  with  $t$ 
7      else                      //execute random-pairing step
8        if  $random() < c_b/(c_b + c_g)$ 
9           $c_b \leftarrow c_b - 1$ 
10         insert  $t$  into  $S$ 
11       else
12          $c_g \leftarrow c_g - 1$ 
13     else                      //a deletion
14       if  $t \in S$ 
15          $c_b \leftarrow c_b + 1$ 
16         remove  $t$  from  $S$ 
17       else
18          $c_g \leftarrow c_g + 1$ 

```

Fig. 7 Random-pairing algorithm (simple version)

The algorithm works as follows. Deletion of an item is handled by removing the item from the sample, if present, and by incrementing the value of c_b or c_g , as appropriate. If $d = 0$, i.e., there are no uncompensated deletions, then insertions are processed as in standard RS. If $d > 0$, then we flip a coin at each insertion step, and include the incoming insertion into the sample with probability $c_b/(c_b + c_g)$; otherwise, we exclude the item from the sample. We then decrease either c_b or c_g , depending on whether the insertion has been included into the sample or not. Conceptually, whenever an item is inserted and $d > 0$, the item is paired with a randomly selected uncompensated deletion, called the “partner” deletion. The inserted item is included into the sample if its partner was in the sample at the time of its deletion, and excluded otherwise. The probability that the partner was in the sample is $c_b/(c_b + c_g)$. For purposes of sample maintenance, it is not necessary to keep track of the precise identity of the random partner; it suffices to maintain the counters c_b and c_g .

Figure 7 displays the pseudocode for the simplest version of the RP algorithm, which is executed whenever a new transaction γ_i arrives. As with classic reservoir sampling, the basic algorithm of Fig. 7 can be speeded up by directly generating random skip values; see [61] for such optimizations, as well as a correctness proof and a technique for merging samples.

Note that, if boundedness of the sample size is not a concern, then the following simple Bernoulli sampling scheme can be used to maintain S . First fix a sampling rate $q \in (0, 1)$. For an insertion transaction $\gamma_i = +t_k$, include t_k in S with probability q and exclude t_k with probability $1 - q$. For a deletion transaction $\gamma_i = -t_k$, simply remove t_k from S , if present.

In a variant of the above setting, multiple copies of each item may occur in both the window W and the sample S , so that both W and S are multisets (i.e., bags). When sampling from multisets, relatively sophisticated techniques are required to handle deletion of items. An extension of Bernoulli sampling to multisets is given in [62], and the authors in [63–65] provide techniques for maintaining a uniform sample D of the distinct items in a multiset window W and, for each item in D , an exact value (or high-precision estimate) of the frequency of the item in W .

4 Inference from a Sample

This section concerns techniques for drawing inferences about the contents of a window from a sample of window elements. As discussed in Sect. 1, such techniques belong to the domain of finite-population sampling. A complete discussion of this topic is well beyond the scope of this chapter, and so we cover only the most basic results; see [1–5] for further discussion. Our emphasis is on methods for estimating population sums and functions of such sums—these fundamental population parameters occur frequently in practice and are well understood. The “population” is, of course, the set of all elements in the window.

4.1 Estimation of Population Sums and Functions of Sums

We first describe techniques for estimating quantities of the form $\theta = \sum_{e_i \in W} h(e_i)$, where W is the window of interest, assumed to be of length n , and h is a real-valued function. We then discuss estimation methods for window characteristics of the form $\alpha = g(\theta_1, \theta_2, \dots, \theta_d)$, where $d \geq 1$. Here $g: \mathfrak{N}^d \mapsto \mathfrak{N}$ is a specified “smooth” function and each θ_j is a population sum, i.e., $\theta_j = \sum_{e_i \in W} h_j(e_i)$ for some function h_j . Some examples of these estimands are as follows, where each element e_i is a sales-transaction record and $v(e_i)$ is the dollar value of the transaction.

1. Let $h(e_i) = v(e_i)$. Then θ is the sum of sales over the transactions in the window.
2. Let h be a predicate function such that $h(e_i) = 1$ if $v(e_i) > \$1000$ and $h(e_i) = 0$ otherwise. Then θ is the total number of transactions in the window that exceed \$1000.
3. Let θ be as in example 1 above, and let $g(\theta) = \theta/n$. Then $\alpha = g(\theta)$ is the average sales amount per transaction in the window. If θ is as in example 2 above, then α is the fraction of transactions that exceed \$1000.

4. Suppose that an element not only records the dollar value of the transaction, but also enough information to compute the relative position number of the element within the window (the element in relative position 1 being the first to have arrived). Let $0 \leq w_1 \leq w_2 \leq \dots \leq w_n$ be a sequence of nondecreasing weights. If $h(e_i) = w_i v(e_i)$, then θ is a weighted sum of sales that favors more recent arrivals.
5. Let $h_1(e_i) = v(e_i)$ if $v(e_i) > \$1000$ and $h_1(e_i) = 0$ otherwise, and let h_2 be as in example 2 above. Also let θ_1 and θ_2 be the population sums that correspond to h_1 and h_2 . If $g(x, y) = x/y$, then $\alpha = g(\theta_1, \theta_2)$ is the average value of those transactions in the window that exceed \$1000.
6. Let $h_1(e_i) = v(e_i)$ and $h_2(e_i) = v^2(e_i)$. Also let $g(x, y) = (y/n) - (x/n)^2$. Then $\alpha = g(\theta_1, \theta_2)$ is the variance of the sales amounts for the transactions in the window.

As discussed in Sect. 1.1, any estimate $\hat{\theta}$ of a quantity such as θ should be supplemented with an assessment of the estimate's accuracy and precision. Typically, $\hat{\theta}$ will be (at least approximately) *unbiased*, in that $E[\hat{\theta}] = \theta$, i.e., if the sampling experiment were to be repeated multiple times, the estimator $\hat{\theta}$ would equal θ on average. In this case, the usual measure of precision is the *standard error*,⁸ defined as the standard deviation, or square root of the variance, of $\hat{\theta}$, $SE[\hat{\theta}] = E^{1/2}[(\hat{\theta} - E[\hat{\theta}])^2]$. If the distribution of $\hat{\theta}$ is approximately normal and we can compute from the sample an estimator $\hat{SE}[\hat{\theta}]$ of $SE[\hat{\theta}]$, then we can go further and make probabilistic statements of the form “with probability approximately $100p\%$, the unknown value θ lies in the (random) interval $[\hat{\theta} - z_p \hat{SE}[\hat{\theta}], \hat{\theta} + z_p \hat{SE}[\hat{\theta}]]$,” where z_p is the $(p+1)/2$ quantile of a standard (mean 0, variance 1) normal distribution. That is, we can compute *confidence intervals* for θ . A number of estimators $\hat{\theta}$, including the SRS-based expansion estimator discussed below, have approximately a normal distribution under mild regularity conditions. These conditions require, roughly speaking, that (i) the window length n be large, (ii) the sample size k be much smaller than n but reasonably large in absolute terms, say, 50 or larger, and (iii) the population sum not be significantly influenced by any one value or small group of values. Formal statements of these results take the form of “finite-population central limit theorems” [4, Sects. 3.4 and 3.5].

4.2 SRS and Bernoulli Sampling

For an SRS S of size k , the standard estimator of a population sum is the obvious one, namely the *expansion estimator* $\hat{\theta} = (n/k) \sum_{e_i \in S} h(e_i)$; cf. the example in Sect. 1.1.

⁸For a biased estimator $\hat{\theta}$, the usual precision measure is the *root mean squared error*, defined as $RMSE[\hat{\theta}] = E^{1/2}[(\hat{\theta} - \theta)^2]$. It is not hard to show that $RMSE = (\text{bias}^2 + SE^2)^{1/2}$, so that RMSE and SE coincide for an unbiased estimator.

Similarly, the estimator of the corresponding population average is the sample average $\hat{\alpha} = \hat{\theta}/n = (1/k) \sum_{e_i \in S} h(e_i)$. Both of these estimators are unbiased and *consistent*, in the sense that they converge to their true values as the sample size increases. For Bernoulli sampling with sampling rate q , the sample size k is random, and there are two possible estimators of the population sum θ : the expansion estimator (with k equal to the observed sample size) and the estimator $\theta^* = (1/q) \sum_{e_i \in S} h(e_i)$. The estimator θ^* is unbiased; the estimator $\hat{\theta}$ is slightly biased,⁹ but often has significantly lower variance than θ^* , and is usually preferred in practice. The variance of $\hat{\theta}$ is given by $\text{Var}[\hat{\theta}] = (n^2/k)(1-f)\sigma^2$ under SRS (exactly) and under Bernoulli sampling (approximately), where $f = k/n$ is the sampling fraction and

$$\sigma^2 = \frac{\sum_{e_i \in W} (h(e_i) - (\theta/n))^2}{n-1} \quad (4)$$

is the variance¹⁰ of the numbers $\{h(e_i) : e_i \in W\}$. An unbiased estimator of $\text{Var}[\hat{\theta}]$, based on the values in the sample, is $\hat{\text{Var}}[\hat{\theta}] = (n^2/k)(1-f)\hat{\sigma}^2$, where $\hat{\sigma}^2 = (1/(k-1)) \sum_{e_i \in S} (h(e_i) - (\hat{\theta}/n))^2$. To obtain corresponding variance formulas for the sample average $\hat{\alpha}$, simply multiply by n^{-2} , i.e., $\text{Var}[\hat{\alpha}] = (\sigma^2/k)(1-f)$ and $\hat{\text{Var}}[\hat{\alpha}] = (\hat{\sigma}^2/k)(1-f)$. To obtain expressions for standard errors and their estimators, take the square root of the corresponding expressions for variances.

4.3 Stratified Sampling

Here we consider the simple stratified sampling scheme for a stationary window discussed in Sect. 2.3. Our results also apply to the stratified sampling scheme for sliding windows in Sect. 3.1, at those times when the window boundaries align with the strata boundaries; the modifications required to deal with arbitrary time points can be derived, e.g., from the results in [1, Chap. 5]. Suppose that our goal is to estimate an unknown population sum θ . Also suppose that there are L equal-sized strata, with $m = n/L$ elements per stratum, and that we obtain an SRS of $r = k/L$ elements from each stratum. Denote by Λ_j the set of elements in the j th stratum and by S_j the elements of Λ_j that are in the final sample. The usual expansion estimator $\hat{\theta}$ is unbiased for θ , and

$$\text{Var}[\hat{\theta}] = m \left(\frac{n}{k} - 1 \right) \sum_{j=1}^L \sigma_j^2,$$

⁹The bias arises from the fact that the sample can be empty, albeit typically with low probability.

¹⁰We follow the survey-sampling literature, which usually takes the denominator in (4) as $n-1$ instead of n because this convention leads to simpler formulas.

where $\sigma_j^2 = (1/(m-1)) \sum_{e_i \in A_j} (h(e_i) - (\theta_j/m))^2$ and $\theta_j = \sum_{e_i \in A_j} h(e_i)$. An unbiased estimator of $\text{Var}[\hat{\theta}]$ is

$$\hat{\text{Var}}[\hat{\theta}] = m \left(\frac{n}{k} - 1 \right) \sum_{j=1}^L \hat{\sigma}_j^2,$$

where $\hat{\sigma}_j^2 = (1/(r-1)) \sum_{e_i \in S_j} (h(e_i) - (\hat{\theta}_j/m))^2$ and $\hat{\theta}_j = (m/r) \sum_{e_i \in S_j} h(e_i)$.

Observe that if the strata are highly homogeneous, then each σ_j^2 is very small, so that $\text{Var}[\hat{\theta}]$ is very small. Indeed, it can be shown [1, Sect. 5.6] that if $m \sum_{j=1}^L ((\theta_j/m) - (\theta/n))^2 \geq (1 - (m/n)) \sum_{j=1}^L \sigma_j^2$, then the variance of $\hat{\theta}$ under stratified sampling is less than or equal to the variance under simple random sampling. This condition holds except when the stratum means are almost equal, i.e., if the data are even slightly stratified, then stratified sampling yields more precise results than SRS.

4.4 Biased Sampling

When using a biased sampling scheme, we can, in principle, recover an unbiased estimate of a population sum by using a *Horvitz–Thompson* (HT) estimator; see, for example, [3], where these estimators are called π -estimators. The general form of an HT-estimator for a population sum of the form $\theta = \sum_{i \in W} h(e_i)$ based on a sample $S \subseteq W$ is $\hat{\theta}_{\text{HT}} = \sum_{i \in S} (h(e_i)/\pi_i)$, where π_i is the probability that element e_i is included in S . Assume that $\pi_i > 0$ for each e_i , and let $\Phi_i = 1$ if $e_i \in S$ and $\Phi_i = 0$ otherwise, so that $E[\Phi_i] = \pi_i$. Observe that

$$E[\hat{\theta}] = E \left[\sum_{i \in W} h(e_i) \Phi_i / \pi_i \right] = \sum_{i \in W} h(e_i) E[\Phi_i] / \pi_i = \theta, \quad (5)$$

so that HT-estimators are indeed unbiased. Similar calculations [3, Result 2.8.1] show that the variance of $\hat{\theta}$ is given by

$$\text{Var}[\hat{\theta}] = \sum_{i \in W} \sum_{j \in W} \left(\frac{\pi_{ij}}{\pi_i \pi_j} - 1 \right) h(e_i) h(e_j),$$

where π_{ij} is the probability that elements e_i and e_j are both included in S . (Take $\pi_{ii} = \pi_i$ for $i \in W$.) When using biased reservoir sampling, for example, the probabilities π_i and π_{ij} can be determined from (1) and (2). (In this case, and in others arising in practice, it can be expensive to compute the π_i 's and π_{ij} 's.) Provided that $\pi_{ij} > 0$ for all i, j , an unbiased estimator of $\text{Var}[\hat{\theta}]$ is given by

$$\hat{\text{Var}}[\hat{\theta}] = \sum_{i \in S} \sum_{j \in S} \left(\frac{1}{\pi_i \pi_j} - \frac{1}{\pi_{ij}} \right) h(e_i) h(e_j).$$

It can be shown [3, Result 2.8.2] that if the sampling scheme is such that the final sample size is deterministic and each π_{ij} is positive, then alternative forms for the variance and variance estimator are given by

$$\text{Var}[\hat{\theta}] = \frac{1}{2} \sum_{i \in W} \sum_{j \in W} (\pi_i \pi_j - \pi_{ij}) \left(\frac{h(e_i)}{\pi_i} - \frac{h(e_j)}{\pi_j} \right)^2$$

and

$$\hat{\text{Var}}[\hat{\theta}] = \frac{1}{2} \sum_{i \in S} \sum_{j \in S} \left(\frac{\pi_i \pi_j}{\pi_{ij}} - 1 \right) \left(\frac{h(e_i)}{\pi_i} - \frac{h(e_j)}{\pi_j} \right)^2.$$

The latter variance estimator is known as the *Yates–Grundy–Sen* estimator; unlike the previous variance estimator, it has the advantage of always being nonnegative if each term $(\pi_i \pi_j / \pi_{ij}) - 1$ is positive (but is only guaranteed to be unbiased for fixed-size sampling schemes). Most of the estimators discussed previously can be viewed as HT-estimators, for example, the SRS-based expansion estimator: from (1) and (2), we have $\pi_i = k/n$ and $\pi_{ij} = k(k-1)/(n(n-1))$ for all i, j .

In general, quantifying the effects of biased sampling on the outcome of a subsequent data analysis can be difficult. When estimating a population sum, however, the foregoing results lead to a clear understanding of the consequences of biased sampling schemes. Specifically, observe that, by (5), the sample sum $\hat{\theta} = \sum_{e_i \in S} h(e_i)$ is, in fact, an unbiased estimator of the *weighted* sum $\theta = \sum_{e_i \in W} \pi_i h(e_i)$.

4.5 Functions of Population Sums

Suppose that we wish to estimate $\alpha = g(\theta)$, where $\theta = (\theta_1, \theta_2, \dots, \theta_d)$ is a vector of population sums corresponding to real-valued functions h_1, h_2, \dots, h_d . We assume that there exists an unbiased and consistent estimator $\hat{\theta}_j$ for each θ_j , and we write $\hat{\theta} = (\hat{\theta}_1, \hat{\theta}_2, \dots, \hat{\theta}_d)$. We also assume that g is continuous and differentiable in a neighborhood of θ and write $\nabla g = (\nabla_1 g, \nabla_2 g, \dots, \nabla_d g)$ for the gradient of g . A straightforward estimate of α is $\hat{\alpha} = g(\hat{\theta})$, i.e., we simply replace each population sum θ_j by its estimate and then apply the function g . The estimator $\hat{\alpha}$ will in general be biased if g is a nonlinear function. For example, Jensen's inequality [50, Sect. 21] implies that $E[\hat{\alpha}] \geq \alpha$ if g is convex and $E[\hat{\alpha}] \leq \alpha$ if g is concave. The bias decreases, however, as the sample size increases and, moreover, $\hat{\alpha}$ is consistent for α since g is continuous. The variance of $\hat{\alpha}$ is difficult to obtain precisely. If the sample size is large, however, so that with high probability $\hat{\theta}$ is close θ , then we can approximate $\text{Var}[\hat{\alpha}]$ by the variance of a linearized version of α obtained by taking a Taylor expansion around the point θ . That is,

$$\text{Var}[\hat{\alpha}] \approx \text{Var} \left[g(\theta) + \sum_{j=1}^d a_j (\hat{\theta}_j - \theta_j) \right] = \sum_{i=1}^d \sum_{j=1}^d a_i a_j \text{Cov}[\hat{\theta}_i, \hat{\theta}_j],$$

where $a_i = \nabla_i g(\theta)$ and $\text{Cov}[\hat{\theta}_i, \hat{\theta}_j]$ denotes the covariance of $\hat{\theta}_i$ and $\hat{\theta}_j$. We estimate $\text{Var}[\hat{\alpha}]$ by $\hat{\text{Var}}[\hat{\alpha}] = \sum_{i=1}^d \sum_{j=1}^d \hat{a}_i \hat{a}_j \hat{\text{Cov}}[\hat{\theta}_i, \hat{\theta}_j]$, where $\hat{a}_i = \nabla_i g(\hat{\theta})$ and $\hat{\text{Cov}}[\hat{\theta}_i, \hat{\theta}_j]$ is an estimate of $\text{Cov}[\hat{\theta}_i, \hat{\theta}_j]$. The exact formula for the covariance estimator depends on the specific sampling scheme and population-sum estimator. Typically, assuming that $\hat{\theta}_1, \hat{\theta}_2, \dots, \hat{\theta}_d$ are computed from the same sample, such formulas are directly analogous to those for the variance. For example, for the expansion estimator under SRS, we have

$$\hat{\text{Cov}}[\hat{\theta}_i, \hat{\theta}_j] = \frac{n^2(1-f)}{k} \frac{1}{k-1} \sum_{e_l \in S} (h_i(e_l) - (\hat{\theta}_i/n))(h_j(e_l) - (\hat{\theta}_j/n)),$$

where $f = k/n$ and $\hat{\theta}_j = (n/k) \sum_{e_i \in S} h_j(e_i)$ for each j . For stratified sampling with L strata of length m ,

$$\hat{\text{Cov}}[\hat{\theta}_i, \hat{\theta}_j] = m \left(\frac{n}{k} - 1 \right) \sum_{q=1}^L \frac{1}{r-1} \sum_{e_l \in S_q} (h_i(e_l) - (\hat{\theta}_{i,q}/m))(h_j(e_l) - (\hat{\theta}_{j,q}/m)),$$

where, as before, each stratum comprises $m = n/L$ elements, S_q is the SRS of size $r = k/L$ from the q th stratum, and $\hat{\theta}_{s,q} = (m/r) \sum_{e_l \in S_q} h_s(e_l)$ for each s and q . In general, we note that for any sampling scheme (possibly biased) such that the size of the sample S is deterministic and each π_{ij} is positive, the linearization approach leads to the following Yates–Grundy–Sen variance estimator:

$$\hat{\text{Var}}[\hat{\alpha}] = \frac{1}{2} \sum_{i \in S} \sum_{j \in S} \left(\frac{\pi_i \pi_j}{\pi_{ij}} - 1 \right) \left(\frac{z_i}{\pi_i} - \frac{z_j}{\pi_j} \right)^2,$$

where $z_l = \sum_{j=1}^L \nabla_j g(\hat{\theta}) h_j(e_l)$ for each $e_l \in S$; see [4, Sect. 4.2.1].

Acknowledgements The author would like to thank P. Brown, M. Datar, and Rainer Gemulla for several helpful discussions. D. Sivakumar suggested the idea underlying the bound α_2 given in Sect. 3.1.

References

1. W.G. Cochran, *Sampling Techniques*, 3rd edn. (Wiley, New York, 1977)
2. L. Kish, *Survey Sampling* (Wiley, New York, 1965)
3. C.E. Särndal, B. Swensson, J. Wretman, *Model Assisted Survey Sampling* (Springer, New York, 1992)
4. M.E. Thompson, *Theory of Sample Surveys* (Chapman & Hall, London, 1997)
5. S.K. Thompson, *Sampling* (Wiley, New York, 2002)
6. B. Efron, R.J. Tibshirani, *An Introduction to the Bootstrap* (Chapman & Hall, New York, 1993)
7. R. Gemulla, W. Lehner, Deferred maintenance of disk-based random samples, in *Proc. EDBT*. Lecture Notes in Computer Science (Springer, Berlin, 2006), pp. 423–441

8. A. Pol, C.M. Jermaine, S. Arumugam, Maintaining very large random samples using the geometric file. VLDB J. **17**(5), 997–1018 (2008)
9. P.G. Brown, P.J. Haas, Techniques for warehousing of sample data, in *Proc. 22nd ICDE* (2006)
10. P.J. Haas, The need for speed: speeding up DB2 using sampling. IDUG Solut. J. **10**, 32–34 (2003)
11. S. Chaudhuri, R. Motwani, V.R. Narasayya, Random sampling for histogram construction: how much is enough? in *Proc. ACM SIGMOD* (1998), pp. 436–447
12. D. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri, Practical skew handling algorithms for parallel joins, in *Proc. 19th VLDB* (1992), pp. 27–40
13. P.J. Haas, C. König, A bi-level Bernoulli scheme for database sampling, in *Proc. ACM SIGMOD* (2004), pp. 275–286
14. W. Hou, G. Ozsoyoglu, B. Taneja, Statistical estimators for relational algebra expressions, in *Proc. Seventh PODS* (1988), pp. 276–287
15. P.J. Haas, J.M. Hellerstein, Ripple joins for online aggregation, in *Proc. ACM SIGMOD* (1999), pp. 287–298
16. S. Acharya, P. Gibbons, V. Poosala, S. Ramaswamy, Join synopses for approximate query answering, in *Proc. ACM SIGMOD* (1999), pp. 275–286
17. S. Acharya, P. Gibbons, V. Poosala, Congressional samples for approximate answering of group-by queries, in *Proc. ACM SIGMOD* (2000), pp. 487–498
18. S. Chaudhuri, G. Das, M. Datar, R. Motwani, V.R. Narasayya, Overcoming limitations of sampling for aggregation queries, in *Proc. Seventeenth ICDE* (2001), pp. 534–542
19. S. Chaudhuri, R. Motwani, V.R. Narasayya, On random sampling over joins, in *Proc. ACM SIGMOD* (1999), pp. 263–274
20. S. Ganguly, P.B. Gibbons, Y. Matias, A. Silberschatz, Bifocal sampling for skew-resistant join size estimation, in *Proc. ACM SIGMOD* (1996), pp. 271–281
21. V. Ganti, M.L. Lee, R. Ramakrishnan, ICICLES: self-tuning samples for approximate query answering, in *Proc. 26th VLDB* (2000), pp. 176–187
22. P.J. Haas, A.N. Swami, Sampling-based selectivity estimation using augmented frequent value statistics, in *Proc. Eleventh ICDE* (1995), pp. 522–531
23. C. Jermaine, Robust estimation with sampling and approximate pre-aggregation, in *Proc. 29th VLDB* (2003), pp. 886–897
24. W. Hou, G. Ozsoyoglu, B. Taneja, Processing aggregate relational queries with hard time constraints, in *Proc. ACM SIGMOD* (1989), pp. 68–77
25. F. Olken, D. Rotem, Simple random sampling from relational databases, in *Proc. 12th VLDB* (1986), pp. 160–169
26. F. Olken, D. Rotem, Random sampling from B^+ trees, in *Proc. 15th VLDB* (1989), pp. 269–277
27. F. Olken, D. Rotem, Maintenance of materialized views of sampling queries, in *Proc. Eighth ICDE* (1992), pp. 632–641
28. F. Olken, D. Rotem, Sampling from spatial databases, in *Proc. Ninth ICDE* (1993), pp. 199–208
29. F. Olken, D. Rotem, P. Xu, Random sampling from hash files, in *Proc. ACM SIGMOD* (1990), pp. 375–386
30. P.J. Haas, J.F. Naughton, S. Seshadri, A.N. Swami, Selectivity and cost estimation for joins based on random sampling. J. Comput. Syst. Sci. **52**, 550–569 (1996)
31. P.J. Haas, J.F. Naughton, A.N. Swami, On the relative cost of sampling for join selectivity estimation, in *Proc. Thirteenth PODS* (1994), pp. 14–24
32. P.J. Haas, A.N. Swami, Sequential sampling procedures for query size estimation, in *Proc. ACM SIGMOD* (1992), pp. 1–11
33. W. Hou, G. Ozsoyoglu, E. Dogdu, Error-constrained COUNT query evaluation in relational databases, in *Proc. ACM SIGMOD* (1991), pp. 278–287
34. R.J. Lipton, J.F. Naughton, Query size estimation by adaptive sampling, in *Proc. Ninth PODS* (1990), pp. 40–46

35. R.J. Lipton, J.F. Naughton, D.A. Schneider, Practical selectivity estimation through adaptive sampling, in *Proc. ACM SIGMOD* (1990), pp. 1–11
36. R.J. Lipton, J.F. Naughton, D.A. Schneider, S. Seshadri, Efficient sampling strategies for relational database operations. *Theor. Comput. Sci.* **116**, 195–226 (1993)
37. K.D. Seppi, J.W. Barnes, C.N. Morris, A Bayesian approach to database query optimization. *ORSA J. Comput.* **5**, 410–419 (1993)
38. J.M. Hellerstein, P.J. Haas, H.J. Wang, Online aggregation, in *Proc. ACM SIGMOD* (1997), pp. 171–182
39. C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, A. Pol, A disk-based join with probabilistic guarantees, in *Proc. ACM SIGMOD* (2005)
40. C.M. Jermaine, S. Arumugam, A. Pol, A. Dobra, Scalable approximate query processing with the DBO engine, in *Proc. ACM SIGMOD* (2007), pp. 725–736
41. C. Jermaine, A. Dobra, A. Pol, S. Joshi, Online estimation for subset-based SQL queries, in *Proc. 31st VLDB* (2005), pp. 745–756
42. G. Luo, C. Ellman, P.J. Haas, J.F. Naughton, A scalable hash ripple join algorithm, in *Proc. ACM SIGMOD* (2002), pp. 252–262
43. A. Pol, C. Jermaine, Relational confidence bounds are easy with the bootstrap, in *Proc. ACM SIGMOD* (2005)
44. P.G. Brown, P.J. Haas, BHUNT: automatic discovery of fuzzy algebraic constraints in relational data, in *Proc. 29th VLDB* (2003), pp. 668–679
45. I.F. Ilyas, V. Markl, P.J. Haas, P.G. Brown, A. Aboulnaga, CORDS: automatic discovery of correlations and soft functional dependencies, in *Proc. ACM SIGMOD* (2004), pp. 647–658
46. P. Brown, P. Haas, J. Myllymaki, H. Pirahesh, B. Reinwald, Y. Sismanis, Toward automated large-scale information integration and discovery, in *Data Management in a Connected World*, ed. by T. Härdler, W. Lehner (Springer, New York, 2005)
47. M. Charikar, S. Chaudhuri, R. Motwani, V.R. Narasayya, Towards estimation error guarantees for distinct values, in *Proc. Nineteenth PODS* (2000), pp. 268–279
48. P.J. Haas, L. Stokes, Estimating the number of classes in a finite population. *J. Am. Stat. Assoc.* **93**, 1475–1487 (1998)
49. M. Wu, C. Jermaine, A Bayesian method for guessing the extreme values in a data set, in *Proc. 33rd VLDB* (2007), pp. 471–482
50. P. Billingsley, *Probability and Measure*, 2nd edn. (Wiley, New York, 1986)
51. A.M. Law, *Simulation Modeling and Analysis*, 4th edn. (McGraw-Hill, New York, 2007)
52. D.E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms* (Addison-Wesley, Reading, 1969)
53. A.I. McLeod, D.R. Bellhouse, A convenient algorithm for drawing a simple random sample. *Appl. Stat.* **32**, 182–184 (1983)
54. J.S. Vitter, Random sampling with a reservoir. *ACM Trans. Math. Softw.* **11**, 37–57 (1985)
55. J.S. Vitter, Faster methods for random sampling. *Commun. ACM* **27**, 703–718 (1984)
56. M.T. Chao, A general purpose unequal probability sampling plan. *Biometrika* **69**, 653–656 (1982)
57. H. Brönnimann, B. Chen, M. Dash, P.J. Haas, Y. Qiao, P. Scheuermann, Efficient data reduction methods for on-line association rule discovery, in *Data Mining: Next Generation Challenges and Future Directions*, ed. by H. Kargupta, A. Joshi, K. Sivakumar, Y. Yesha (AAAI Press, Menlo Park, 2004)
58. B. Babcock, M. Datar, R. Motwani, Sampling from a moving window over streaming data, in *Proc. 13th SODA* (2002), pp. 633–634
59. T. Hagerup, C. Rub, A guided tour of Chernoff bounds. *Inf. Process. Lett.* **33**, 305–308 (1990)
60. W. Feller, *An Introduction to Probability Theory and Its Applications*, 3rd edn., vol. 1 (Wiley, New York, 1968)
61. R. Gemulla, W. Lehner, P.J. Haas, Maintaining bounded-size sample synopses of evolving datasets. *VLDB J.* **17**(2), 173–202 (2008)
62. R. Gemulla, W. Lehner, P.J. Haas, Maintaining Bernoulli samples over evolving multisets, in *Proc. Twenty Sixth PODS* (2007), pp. 93–102

63. G. Cormode, S. Muthukrishnan, I. Rozenbaum, Summarizing and mining inverse distributions on data streams via dynamic inverse sampling, in *Proc. 31st VLDB* (2005), pp. 25–36
64. G. Frahling, P. Indyk, C. Sohler, Sampling in dynamic data streams and applications, in *Proc. 21st ACM Symp. Comput. Geom.* (2005), pp. 142–149
65. P.B. Gibbons, Distinct sampling for highly-accurate answers to distinct values queries and event reports, in *Proc. 27th VLDB* (2001), pp. 541–550

Quantiles and Equi-depth Histograms over Streams

Michael B. Greenwald and Sanjeev Khanna

1 Introduction

A quantile query over a set S of size n takes as input a quantile ϕ , $0 < \phi \leq 1$, and returns a value $v \in S$ whose rank in the sorted S is ϕn . Computing the median, the 99-percentile, or the quartiles of a set are examples of quantile queries. Many database optimization problems involve approximate quantile computations over large data sets. Query optimizers use quantile estimates to estimate the size of intermediate results and choose an efficient plan among a set of competing plans. Load balancing in parallel databases can be done by using quantile estimates. Above all, quantile estimates can give a meaningful summary of a large data set using a very small memory footprint. For instance, given any data set, one can create a data structure containing 50 observations that can answer any quantile query to within 1 % precision in rank.

Based on the underlying application domain, a number of desirable properties can be identified for quantile computation. In this survey, we will focus on the following three properties: (a) space used by the algorithm; (b) guaranteed accuracy to within a pre-specified precision; and (c) number of passes made.

It is desirable to compute quantiles using the smallest memory footprint possible. We can achieve this by dynamically storing, at any point in time, only a summary of the data seen so far, and not the entire data set. The size and form of such summaries

M.B. Greenwald
Arista Networks, Inc., 5453 Great America Parkway, Santa Clara, CA 95054, USA
e-mail: greenwald@cis.upenn.edu

S. Khanna (✉)
Dept. of Computer and Info. Science, University of Pennsylvania, 3330 Walnut Street,
Philadelphia, PA 19104, USA
e-mail: sanjeev@cis.upenn.edu

are determined by our *a priori* knowledge of the types of quantile queries we expect to be able to answer. We may know in advance that the client intends to ask for a single, specific quantile. Such a single quantile summary is parameterized in advance by the quantile, ϕ , and a desired precision, ϵ . For any $0 < \phi \leq 1$, and $0 \leq \epsilon \leq 1$, an ϵ -approximate ϕ -quantile on a data set of size n is any value v whose rank, $r^*(v)$, is guaranteed to lie between $n(\phi - \epsilon)$ and $n(\phi + \epsilon)$. For example, a 0.01-approximate 0.5-quantile is any value whose rank is within 1 % of the median.

Alternatively, we may know that the client is interested in a range of equally-spaced quantile queries. In such cases we summarize the data by an *equi-depth histogram*. An equi-depth histogram is parameterized by a bucket size, ϕ , and a precision, ϵ . The client may request any, or all, ϕ -quantiles, that is, elements of ranks, $\phi n, 2\phi n, \dots, n$. We say that $H(\phi, \epsilon)$ is an ϵ -approximate equi-depth histogram with bucket width ϕ if for any $i = 1$ to $1/\phi$, it returns a value $v_{i\phi}$ for the $i\phi$ quantile, where $n(i\phi - \epsilon) \leq r^*(v_{i\phi}) \leq n(i\phi + \epsilon)$.

Finally, we may have no prior knowledge of the anticipated queries. Such ϵ -approximate quantile summaries are parameterized only by a desired precision, ϵ . We say that a quantile summary $Q(\epsilon)$ is ϵ -approximate if it can be used to answer any quantile query to within a precision of ϵn . There is a close relation between equi-depth histograms and quantile summaries. $Q(\epsilon)$ can serve as an equi-depth histogram $H(\phi, \epsilon)$ for any $0 < \phi \leq 1$. Conversely, an equi-depth histogram $H(\phi, \epsilon)$ is a ϕ -approximate quantile summary, provided only that $\epsilon \leq \phi/2$.

The rest of this chapter is organized as follows. In Sect. 2, we formally introduce the notion of an approximate quantile summary, and some simple operations that we will use to describe various algorithms for maintaining quantile summaries. Section 3 describes deterministic algorithms for exact selection and for computing approximate quantile summaries. These algorithms give worst-case deterministic guarantees on the accuracy of the quantile summary. In contrast, Sect. 4 describes algorithms with probabilistic guarantees on the accuracy of the summary.

2 Preliminaries

We will assume throughout that the data is presented on a read-only tape where the tape head moves to the right after each unit of time. Each move of the tape head reveals the next observation (element) in the sequence stored on the tape. For convenience, we will simply say that a new observation arrives after each unit of time. We will use n to denote both the number of observations (elements of the data sequence) that have been seen so far as well as the current time. Almost all results presented here concern algorithms that make a single pass on the data sequence. In a multi-pass algorithm, we assume that at the beginning of each pass, the tape head is reset to the left-most cell on the tape. We assume that our algorithms operate in an RAM model of computation. The space $s(n)$ used by an algorithm is measured in terms of the maximum number of words used by an algorithm while processing an input sequence of length n . This model assumes that a single word can store

$\max\{n, |v^*|\}$ where v^* is the observation with largest absolute value that appears in the data sequence.

The set-up as described above concerns an “insertion-only” model that assumes that an observation once presented is not removed at a later time from the data sequence. This is referred to as the *cash register* model in the literature [7]. A more general setting is the *turnstile* model [17] that also allows for deletion of observations. In Sect. 5, we will consider algorithms for this more general setting as well. We note here that a simple modification of the model above can be used to capture the turnstile case: the i th cell on the read-only tape contains both the i th element in the data sequence and an additional bit that indicates whether the element is being inserted or deleted.

An order-statistic query over a data set S takes as input an integer $r \in [1..|S|]$ and outputs an element of rank r in S . We say that the order-statistic query is answered with ϵ -accuracy if the output element is guaranteed to have rank within $r \pm \epsilon n$. For simplicity, we will assume throughout that ϵn is an integer. If $1/\epsilon$ is an integer, then this is easily enforced by batching observations $1/\epsilon$ at a time. If $1/\epsilon$ is not an integer, then let i be an integer such that $1/2^{i+1} < \epsilon < 1/2^i$. We can then replace the ϵ -accuracy requirement by $\epsilon' = 1/2^{i+1}$ which is within a factor of two of the original requirement.

2.1 Quantile Summary

Following [10], we define a *quantile summary* for a set S to be an ordered set $Q = \{q_1, q_2, \dots, q_\ell\}$ along with two functions rmin_Q and rmax_Q such that

- (i) $q_1 \leq q_2 \leq \dots \leq q_\ell$ and $q_i \in S$ for $1 \leq i \leq \ell$.
- (ii) For $1 \leq i \leq \ell$, each q_i has rank at least $\text{rmin}_Q(q_i)$, and at most $\text{rmax}_Q(q_i)$ in S .
- (iii) Finally, q_1 and q_ℓ are the smallest and the largest elements, respectively, in the set S , that is, $\text{rmin}_Q(q_1) = \text{rmax}_Q(q_1) = 1$, and $\text{rmin}_Q(q_\ell) = \text{rmax}_Q(q_\ell) = |S|$.

We will say that Q is a *relaxed quantile summary* if it satisfies properties (i) and (ii) above, and the following relaxation of property (iii): $\text{rmax}_Q(q_1) \leq \epsilon|S|$ and $\text{rmin}_Q(q_\ell) \geq (1 - \epsilon)|S|$.

We say that a summary Q is an ϵ -approximate quantile summary for a set S if it can be used to answer any order statistic query over S with ϵ -accuracy. That is, it can be used to compute the desired order-statistic within a rank error of at most $\epsilon|S|$. The proposition below describes a sufficient condition on the function rmin_Q and rmax_Q to ensure an ϵ -approximate summary.

Proposition 1 ([10]) *Let Q be a relaxed quantile summary such that it satisfies the condition $\max_{1 \leq i < \ell} (\text{rmax}_Q(q_{i+1}) - \text{rmin}_Q(q_i)) \leq 2\epsilon|S|$. Then Q is an ϵ -approximate summary.*

Proof Let $r = \lceil \phi |S| \rceil$. We will identify an index i such that $r - \epsilon |S| \leq \text{rmin}_Q(q_i)$ and $\text{rmax}_Q(q_i) \leq r + \epsilon |S|$. Clearly, such a value q_i approximates the ϕ -quantile to within the claimed error bounds. We now argue that such an index i must always exist.

Let $e = \max_i (\text{rmax}_Q(q_{i+1}) - \text{rmin}_Q(q_i)) / 2$. Consider first the case $r \geq |S| - e$. We have $\text{rmin}_Q(q_\ell) \geq (1 - \epsilon) |S|$, and therefore $i = \ell$ has the desired property. We now focus on the case $r < |S| - e$, and start by choosing the smallest index j such that $\text{rmax}_Q(q_j) > r + e$. If $j = 1$, then j is the desired index since $r + e < \text{rmax}_Q(q_1) \leq \epsilon |S|$. Otherwise, $j \geq 2$, and it follows that $r - e \leq \text{rmin}_Q(q_{j-1})$. If $r - e > \text{rmin}_Q(q_{j-1})$ then $\text{rmax}_Q(q_j) - \text{rmin}_Q(q_{j-1}) > 2e$; a contradiction since $e = \max_i (\text{rmax}_Q(q_{i+1}) - \text{rmin}_Q(q_i)) / 2$. By our choice of j , we have $\text{rmax}_Q(q_{j-1}) \leq r + e$. Thus $i = j - 1$ is an index i with the above described property. \square

In what follows, whenever we refer to a (relaxed) quantile summary as ϵ -approximate, we assume that it satisfies the conditions of Proposition 1.

2.2 Operations

We now describe two operations that produce new quantile summaries from existing summaries, and compute bounds on the precision of the resulting summaries.

The Combine Operation

Let $Q' = \{x_1, x_2, \dots, x_a\}$ and $Q'' = \{y_1, y_2, \dots, y_b\}$ be two quantile summaries. The operation $\text{combine}(Q', Q'')$ produces a new quantile summary $Q = \{z_1, z_2, \dots, z_{a+b}\}$ by simply sorting the union of the elements in two summaries, and defining new rank functions for each element as follows; w.l.o.g. assume that z_i corresponds to some element x_r in Q' . Let y_s be the largest element in Q'' that is not larger than x_r (y_s is undefined if there is no such element), and let y_t be the smallest element in Q'' that is not smaller than x_r (y_t is undefined if there is no such element). Then

$$\begin{aligned} \text{rmin}_Q(z_i) &= \begin{cases} \text{rmin}_{Q'}(x_r) & \text{if } y_s \text{ undefined;} \\ \text{rmin}_{Q'}(x_r) + \text{rmin}_{Q''}(y_s) & \text{otherwise,} \end{cases} \\ \text{rmax}_Q(z_i) &= \begin{cases} \text{rmax}_{Q'}(x_r) + \text{rmax}_{Q''}(y_s) & \text{if } y_t \text{ undefined;} \\ \text{rmax}_{Q'}(x_r) + \text{rmax}_{Q''}(y_t) - 1 & \text{otherwise.} \end{cases} \end{aligned}$$

Lemma 1 *Let Q' be an ϵ' -approximate quantile summary for a multiset S' , and let Q'' be an ϵ'' -approximate quantile summary for a multiset S'' . Then*

`combine`(Q' , Q'') produces an $\bar{\epsilon}$ -approximate quantile summary Q for the multiset $S = S' \cup S''$ where $\bar{\epsilon} = \frac{n'\epsilon' + n''\epsilon''}{n' + n''} \leq \max\{\epsilon', \epsilon''\}$. Moreover, the number of elements in the combined summary is equal to the sum of the number of elements in Q' and Q'' .

Proof Let n' and n'' respectively denote the number of observations covered by Q' and Q'' . Consider any two consecutive elements z_i, z_{i+1} in Q . By Proposition 1, it suffices to show that $\text{rmax}_Q(z_{i+1}) - \text{rmin}_Q(z_i) \leq 2\bar{\epsilon}(n' + n'')$. We analyze two cases. First, z_i, z_{i+1} both come from a single summary, say elements x_r, x_{r+1} in Q' . Let y_s be the largest element in Q'' that is smaller than x_r and let y_t be the smallest element in Q'' that is larger than x_{r+1} . Observe that if y_s and y_t are both defined, then they must be consecutive elements in Q'' :

$$\begin{aligned} \text{rmax}_Q(z_{i+1}) - \text{rmin}_Q(z_i) &\leq [\text{rmax}_{Q'}(x_{r+1}) + \text{rmax}_{Q''}(y_t) - 1] \\ &\quad - [\text{rmin}_{Q'}(x_r) + \text{rmin}_{Q''}(y_s)] \\ &\leq [\text{rmax}_{Q'}(x_{r+1}) - \text{rmin}_{Q'}(x_r)] \\ &\quad + [\text{rmax}_{Q''}(y_t) - \text{rmin}_{Q''}(y_s) - 1] \\ &\leq 2(n'\epsilon' + n''\epsilon'') = 2\bar{\epsilon}(n' + n''). \end{aligned}$$

Otherwise, if only y_s is defined, then it must be the largest element in Q'' ; or if only y_t is defined, it must be the smallest element in Q'' . A similar analysis can be applied for both these cases as well.

Next we consider the case when z_i and z_{i+1} come from different summaries, say, z_i corresponds to x_r in Q' and z_{i+1} corresponds to y_t in Q'' . Then observe that x_r is the largest element smaller than y_t in Q' and that y_t is the smallest element larger than x_r in Q'' . Moreover, x_{r+1} is the smallest element in Q' that is larger than y_t , and y_{t-1} is the largest element in Q'' that is smaller than x_r . Using these observations, we get

$$\begin{aligned} \text{rmax}_Q(z_{i+1}) - \text{rmin}_Q(z_i) &\leq [\text{rmax}_{Q''}(y_t) + \text{rmax}_{Q'}(x_{r+1}) - 1] \\ &\quad - [\text{rmin}_{Q'}(x_r) + \text{rmin}_{Q''}(y_{t-1})] \\ &\leq [\text{rmax}_{Q''}(y_t) - \text{rmin}_{Q''}(y_{t-1})] \\ &\quad + [\text{rmax}_{Q'}(x_{r+1}) - \text{rmin}_{Q'}(x_r) - 1] \\ &\leq 2(n'\epsilon' + n''\epsilon'') = 2\bar{\epsilon}(n' + n''). \quad \square \end{aligned}$$

Corollary 1 Let Q be a quantile summary produced by repeatedly applying the `combine` operation to an initial set of summaries $\{Q_1, Q_2, \dots, Q_q\}$ such that Q_i is an ϵ_i -approximate summary. Then regardless of the sequence in which `combine` operations are applied, the resulting summary Q is guaranteed to be $(\max_{i=1}^q \epsilon_i)$ -approximate.

Proof By induction on q . The base case of $q = 2$ follows from Lemma 1. Otherwise, $q > 2$, and we can partition the set of indices $I = \{1, 2, \dots, q\}$ into two disjoint sets I_1 and I_2 such that Q is a result of the `combine` operation applied to summary Q' resulting from a repeated application of `combine` to $\{Q_i | i \in I_1\}$, and summary Q'' results from a repeated application of `combine` to $\{Q_i | i \in I_2\}$. By induction hypothesis, Q' is $\max_{i \in I_1} \epsilon_i$ -approximate and Q'' is $\max_{i \in I_2} \epsilon_i$ -approximate. By Lemma 1, then Q must be $\max_{i \in I_1 \cup I_2} \epsilon_i = \max_{i \in I} \epsilon_i$ -approximate. \square

The Prune Operation

The prune operation takes as input an ϵ' -approximate quantile summary Q' and a parameter K , and returns a new summary Q of size at most $K + 1$ such that Q is an $(\epsilon' + (1/(2K)))$ -approximate quantile summary for S . Thus `prune` trades off slightly on accuracy for potentially much reduced space. We generate Q by querying Q' for elements of rank $1, |S|/K, 2|S|/K, \dots, |S|$, and for each element $q_i \in Q$, we define $\text{rmin}_Q(q_i) = \text{rmin}_{Q'}(q_i)$, and $\text{rmax}_Q(q_i) = \text{rmax}_{Q'}(q_i)$.

Lemma 2 *Let Q' be an ϵ' -approximate quantile summary for a multiset S . Then `prune`(Q', K) produces an $(\epsilon' + 1/(2K))$ -approximate quantile summary Q for S containing at most $K + 1$ elements.*

Proof For any pair of consecutive elements q_i, q_{i+1} in Q , $\text{rmax}_Q(q_{i+1}) - \text{rmin}_Q(q_i) \leq (\frac{1}{K} + 2\epsilon')|S|$. By Proposition 1, it follows that Q must be $(\epsilon' + 1/(2K))$ -approximate. \square

3 Deterministic Algorithms

In this section, we will develop a unified framework that captures many of the known deterministic algorithms for computing approximate quantile summaries. This framework appeared in the work of Manku, Rajagopalan, and Lindsay [14], and we refer to it as the MRL framework. We show various earlier approaches for computing approximate quantile summaries are all captured by this framework, and the best-possible algorithm in this framework computes an ϵ -approximate quantile summary using $O(\log^2(\epsilon n)/\epsilon)$ space. We then present an algorithm due to Greenwald and Khanna [10] that deviates from this framework and reduces the space needed to $O(\log(\epsilon n)/\epsilon)$. This is the current best known bound on the space needed for computing an ϵ -approximate quantile summary. We start with some classical results on exact algorithms for selection.

3.1 Exact Selection

In a natural but a restricted model of computation, Munro and Patterson [16] established almost tight bounds on deterministic selection with bounded space. In their model, the only operation that is allowed on the underlying elements is a pairwise comparison. At any time, the summary stores a subset of the elements in the data stream. They considered multi-pass algorithms and showed that any *comparison-based* algorithm that solves the selection problem in p passes requires $\Omega(n^{1/p})$ space. Moreover, there is a simple algorithm that can solve the selection problem in p passes using only $O(n^{1/p}(\log n)^{2-2/p})$ space. We will sketch here the proofs of both these results. We start with the lower bound result.

We focus on the problem of determining the median element using space s . Fix any deterministic algorithm and let us consider the first pass made by the algorithm. Without any loss of generality, we may assume that the first s elements seen by the algorithm get stored in the summary Q . Now each time an element x is brought into Q , some element y is evicted from the summary Q . Let $U(y)$ denote the set of elements that were evicted from Q to make room for element y directly or indirectly. An element z is *indirectly evicted* by an element y in Q if the element y' evicted to make room for y directly or indirectly evicted the element z . Clearly, x will never get compared to any elements in $U(y)$ or the element y . We set $U(x) = U(y) \cup \{y\}$. The adversary now ensures that x is indistinguishable from any element in $U(x)$ with respect to the elements seen so far. Let z_1, z_2, \dots, z_s be the elements in Q after the first $n/2$ elements have been seen. Then $\sum_{i=1}^s |U(z_i)| = n/2 - s$, and by the pigeonhole principle, there exists an element $z_j \in Q$ such that $|z_j \cup U(z_j)| \geq n/(2s)$. The adversary now adjusts the values of the remaining $n/2$ elements so as to ensure that the median element for the entire sequence is the median element of the set $U(z_j) \cup \{z_j\}$. Thus after one pass, the problem size reduces by a factor of $2s$ at most. For the algorithm to succeed in p passes, we must have $(2s)^p \geq n$, that is, $s = \Omega(n^{1/p})$.

Theorem 1 ([16]) *Any p -pass comparison-based algorithm to solve the selection problem on a stream of n elements requires $\Omega(n^{1/p})$ space.*

Recently, Guha and McGregor [12], using techniques from communication complexity, have shown that *any* p -pass algorithm to solve the selection problem on a stream of n elements requires $\Omega(n^{1/p}/p^6)$ bits of space.

The Munro and Patterson algorithm that almost achieves the space bound given in Theorem 1 proceeds as follows. The algorithm maintains at all times a left and a right “filter” such that the desired element is guaranteed to lie between them. At the beginning, the left filter is assumed to be $-\infty$ and the right filter is assumed to be $+\infty$. Starting with an initial bound of n candidate elements contained between the left and the right filters, the algorithm in each pass gradually tightens the gap between the filters until the final pass where it is guaranteed to be less than s . The final pass is then used to determine the exact rank of the filters and retain all candidates in between to output the appropriate answer to the selection problem. The key property that is at the core of their algorithm is as follows.

Lemma 3 *If at the beginning of a pass, there are at most k elements that can lie between the left and the right filters, then at the end of the pass, this number reduces to $O((k \log^2 k)/s)$.*

Thus each pass of the algorithm may be viewed as an approximate selection step, with each step refining the range of the approximation achieved by the preceding step. We describe the precise algorithmic procedure to achieve this in the next subsection. Assuming the lemma, it is easy to see that by choosing $s = \Theta(n^{1/p}(\log n)^{2-2/p})$, we can ensure that after the i th pass, the number of candidate elements between the filters reduces to at most $n^{\frac{p-i}{p}} \log^{\frac{2i}{p}} n$. Setting $i = p - 1$, ensures that the number of candidate elements in the p th pass is at most $n^{1/p}(\log n)^{2-2/p}$.

3.2 MRL Framework for ϵ -Approximate Quantile Summaries

A natural way to construct quantile summaries of large quantities of data is by merging several summaries of smaller quantities of data. Manku, Rajagopalan, and Lindsay [14] noted that all one-pass approximate quantile algorithms prior to their work (most notably, [1, 16]) fit this pattern. They defined a framework, referred from here on as the *MRL framework*, in which all algorithms can be expressed in terms of two basic operations on existing quantile summaries, NEW and COLLAPSE. Each algorithm in the framework builds a quantile summary by applying these operations to members of a set of smaller, fixed size, quantile summaries. These fixed size summaries are referred to as *buffers*. A buffer is a quantile summary of size k that summarizes a certain number of observations. When a buffer summarizes k' observations we define the *weight* of the buffer to be $\lceil \frac{k'}{k} \rceil$.

NEW fills a buffer with k new observations from the input stream (we assume that n is always an integer multiple of k). COLLAPSE takes a set of buffers as input and returns a single buffer summarizing all the input buffers.

Each algorithm in the framework is parameterized by b , the total number of buffers, and k , the number of entries per buffer, needed to summarize a sample of size n to precision ϵ , as well as a *policy* that determines when to apply NEW and COLLAPSE. Further, the authors of [14] proposed a new algorithm that improved upon the space bk needed to summarize n observations to a given precision ϵ . In light of more recent work, it is illuminating to recast the MRL framework in terms of rmin_Q and rmax_Q for each entry in the buffer.

A buffer of weight w in the MRL framework is a quantile summary of kw observations, where k is the number of (sorted) entries in the buffer. Each entry in the buffer consists of a single value that represents w observations in the original data stream. We associate a level, l , with each buffer. Buffers created by NEW have a level of 0, and buffers created by COLLAPSE have a level, l' , that is one greater than the maximum l of the constituent buffers.

NEW takes the next k observations from the input stream, sorts them in ascending order, and stores them in a buffer, setting the weight of this new buffer to 1. This buffer can reproduce the entire sequence of k observations and therefore rmin and rmax of the i th element both equal i , and the buffer has precision that can satisfy even $\epsilon = 0$.

COLLAPSE summarizes a set of α buffers, $B_1, B_2, \dots, B_\alpha$, with a single buffer B , by first calling $\text{combine}(B_1, B_2, \dots, B_\alpha)$, and then¹ $\text{prune}(B, k - 1)$. The weight of this new buffer is $\sum_j w_j$.

Lemma 4 *Let B be a buffer created by invoking COLLAPSE on a set of α buffers, $B_1, B_2, \dots, B_\alpha$, where each buffer B_i has weight w_i and precision ϵ_i . Then B has precision $\leq 1/(2k - 2) + \max_i\{\epsilon_i\}$.*

Proof By Corollary 1, repeated application of combine creates a temporary summary, Q , with precision $\max_i\{\epsilon_i\}$ and αk entries. By Lemma 2, $B = \text{prune}(Q, k - 1)$ produces a summary with an ϵ that is $1/(2k - 2)$ more than the precision of Q . \square

We can view the execution of an algorithm in this framework as a tree. Each node represents the creation of a new buffer of size k : leaves represent NEW operations and internal nodes represent COLLAPSE. Figure 1 represents an example of such a tree. The number next to each node specifies the weight of the resulting buffer. The level, l , of a buffer represents its height in this tree.

Lemma 4 shows that each COLLAPSE operation adds at most $1/(2k)$ to the precision of the buffer. It follows from repeated application of Lemma 4 that a buffer of level l has a precision of $l/(2k)$. Similarly, we can relate the precision of the final summary to the height of the tree.

Corollary 2 *Let $h(n)$ denote the maximum height of the algorithm tree on an input stream of n elements. Then the final summary produced by the algorithm is $(h(n)/(2k))$ -approximate.*

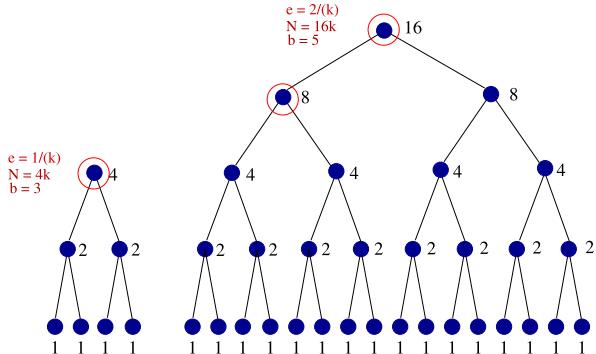
We now apply the lemmas to several different algorithms, in order to compute the space requirements for a given precision and a given number of observations we wish to summarize.

The Munro–Patterson Algorithm

The Munro–Patterson algorithm [16] initially allocates b empty buffers. After k new observations arrive, if an empty buffer exists, then NEW is invoked. If no empty buffer exists, then it creates an empty buffer by calling COLLAPSE on two buffers of

¹The prune phase of COLLAPSE in the MRL paper differs very slightly from our prune .

Fig. 1 Tree representations of Munro–Patterson algorithm for $b = 3$, and 5. Note that the final state of the algorithm under a root consists of the pair of buffers that are the children of the root. The shape and height of a tree depends only on b , and is independent of k , but the precision, ϵ , and the number of observations summarized, n , are both functions of k



equal weight. Figure 1 represents the Munro–Patterson algorithm for small values of b .

Let $h = \lceil \log(n/k) \rceil$. Since the algorithm merges at each step buffers of equal weight, it follows that the resulting tree is a balanced binary tree of height h where the leaves represent k observations each, and each internal node corresponds to $k2^i$ observations for some integer $1 \leq i \leq h$. The number of available buffers b must satisfy the constraint $b \geq h$ since if $n = k2^h - 1$, the resulting summary requires h buffers with distinct weights of 1, 2, 4, etc. By Corollary 2, the resulting summary is guaranteed to be $h/2k$ -approximate. Given a desired precision ϵ , we need to satisfy $h/2k \leq \epsilon$. It is easy to verify that choosing $k = \lceil (\log(2\epsilon n))/(2\epsilon) \rceil$ satisfies the precision requirement. Thus the total space used by this algorithm is $bk = O(\log^2(\epsilon n)/\epsilon)$.

The Alsabti–Ranka–Singh Algorithm

The Alsabti–Ranka–Singh Algorithm [1] allocates b buffers and divides them, equally, into two classes. The first $b/2$ buffers are reserved for the leaves of the tree. Each group of $kb/2$ observations are collected into $b/2$ buffers using NEW, and then COLLAPSEd into a single buffer from the second class. This process is repeated $b/2$ times, resulting in $b/2$ buffers with weight $b/2$ as children of the root. After the last such operation, the $b/2$ leaf buffers are discarded.

The depth of the Alsabti–Ranka–Singh tree (see Fig. 2) is always 2, so by Corollary 2, $k \geq 1/\epsilon$. We need $k(b/2)^2 \geq n$ to cover all the observations. Given that b increases coverage quadratically and k only linearly, it is most efficient to choose the largest b and smallest k that satisfy the above constraints if we wish to minimize bk . The smallest k is $1/\epsilon$, hence $(b/2)^2 \geq \epsilon n$, so $b \geq \sqrt{\epsilon n}/4$, and $bk = O(\sqrt{n/\epsilon})$.

The Manku–Rajagopalan–Lindsay Algorithm

It is natural to try to devise the best algorithm possible within the MRL framework. It is easy to see that, for a given b and k , the more leaves an algorithm tree has,

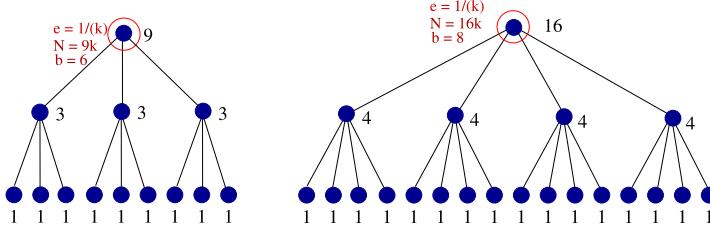


Fig. 2 Tree representation of Alsabti–Ranka–Singh algorithm. For any specific choice of b_1 and b_2 , for $b_1 \neq b_2$, the tree for $b = b_1$ is not a subtree of $b = b_2$. The precision, ϵ , and the number of observations summarized, n , are both functions of k

the more observations it summarizes. Also, from Corollary 2, the shallower the tree, the more precise the summary is. Clearly, for a fixed b it is best to construct the shallowest and widest tree possible, in order to summarize the most observations with the finest precision.

However, both algorithms presented above are inefficient in this light. For example, Alsabti–Ranka–Singh is not as wide as possible. After the algorithm fills the first $b/2$ buffers, it invokes COLLAPSE, leaving all buffers empty except for one buffer with precision $1/(2k)$ summarizing $bk/2$ observations. However, there is no need for it to call COLLAPSE at that point—there are $b/2$ empty buffers remaining. If it deferred calling COLLAPSE until after filling all b buffers, the results would again be all buffers empty except for one buffer with precision $1/(2k)$, but this time summarizing bk observations. Even worse, after $b/2$ calls to COLLAPSE, Alsabti–Ranka–Singh discards the $b/2$ “leaf buffers”, although if it kept those buffers, and continued collecting, it could keep on collecting, roughly, a factor of $b/2$ times as many observations with no loss of precision.

The Munro–Patterson algorithm *does* use empty buffers greedily. However, it is not as shallow as possible. Munro–Patterson requires a tree of height $\log \beta$ to combine β buffers because it only COLLAPSES pairs of buffers at a time instead of combining the entire set at once. Had Munro–Patterson called COLLAPSE on the entire set in a single operation, it would end with a buffer with $\log \beta/(2k)$ higher precision (there is a loss of precision of $1/(2k)$ for each call to COLLAPSE).

The new Manku–Rajagopalan–Lindsay (MRL) algorithm [14] aims to use the buffers as efficiently as possible—to build the shallowest, widest tree it can for a fixed b . The MRL algorithm never discards buffers; it uses any buffers that are available to record new observations. The basic approach taken by MRL is to keep the algorithm tree as wide as possible. It achieves this by labeling each buffer B_j with a level L_j , which denotes its height (see Fig. 3). Let l denote the smallest value of L_j for all existing, full, buffers. The MRL policy is to allocate new buffers at level 0 until the buffer pool is exhausted, and then to call COLLAPSE on all buffers of level l . More specifically, MRL considers two cases:

- Empty buffers exist. Call NEW on each and assign level 0 to them.
- No empty buffers exist. Call COLLAPSE on all buffers of level l and assign the output buffer a level L of $l + 1$.

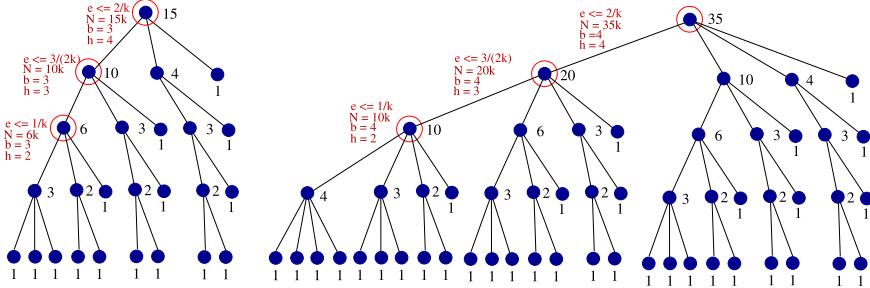


Fig. 3 Tree representation of Manku–Rajagopalan–Lindsay algorithm

If level l contains only 2 buffers, then COLLAPSE frees only a single buffer which NEW assigns level 0. When that buffer is filled, it is the only buffer at level 0. Calling COLLAPSE on a single buffer merely increments the level without modifying the buffer. This will continue until the buffer is promoted to level l , where other buffers exist. Thus MRL treats a third case specially:

- Precisely one empty buffer exists. Call NEW on it and assign it level l .

Proposition 2 *In the tree representing the COLLAPSES and NEWS in an MRL algorithm with b buffers, the number of leaves in a subtree of height h , $L(b, h)$, is $\binom{b+h-2}{h-1}$.*

Proof We will prove by induction on h that $L(b, h) = \binom{(b-1)+(h-1)}{h-1}$.

For $h = 1$ the tree is a single node, a leaf. So, for all b , $L(b, 1) = 1 = \binom{b-1}{0}$.

Assume that for all $h' < h$, for all b , that $L(b, h') = \binom{(b-1)+(h'-1)}{h'-1}$.

$L(b, h)$ is equal to $\sum_{i=1}^b L(i, h - 1)$. To see this, note that we build the h th level by finishing a tree of $(b, h - 1)$, then collapsing it all into 1 buffer. Now we have $b - 1$ buffers left over to build another tree of height $h - 1$. When we finish, we collapse *that* into a single buffer, and start over building a tree of height $h - 1$ with $b - 2$ buffers, and so on, until we are left with only 1 buffer, which we fill. At that point we have no free buffers left and so we collapse all b buffers into the single buffer that is the root at height h . By the induction hypothesis, we know that $L(b, h - 1) = \binom{(b-1)+(h-2)}{h-2}$. Therefore, $L(b, h) = \sum_{i=1}^b \binom{(i-1)+(h-2)}{h-2}$, or $L(b, h) = \sum_{i=0}^{b-1} \binom{(i+(h-2))}{h-2}$. But by summation on the upper index, we have $L(b, h) = \sum_{i=0}^{b-1} \binom{(i+(h-2))}{h-2} = \binom{(b-1)+1+h-2}{h-1} = \binom{(b-1)+(h-1)}{h-1}$. \square

The leaf buffers must include all n observations, so by Proposition 2, b, k , and h must be chosen such that $kL(b, h) = k \binom{b+h-2}{h-1} \geq n$. The summary must be ϵ -approximate, so by Corollary 2, $h/(2k) \geq \epsilon'$. We must choose b, k , and h to satisfy these constraints while minimizing bk . Increasing h (up to the point we would violate the precision requirement) increases space-efficiency because larger h cov-

ers more observations without increasing the memory footprint, bk . The largest h that bounds the precision of the summary to be within ϵ , is $h = 2\epsilon k$.

Now that h can be computed as a function of k and ϵ , we can focus on choosing the best values of b and k to minimize the space bk required. We first show that if a pair b, k is *space-efficient*—meaning that no other pair b', k' could cover more observations in the same space bk —then $k = O(b/\epsilon)$.

The number of observations covered by the MRL algorithm for a pair b, k is $kL(b, 2\epsilon k)$. $L(b, h)$ is symmetric in b and h (e.g., $L(b, h) = L(h, b)$). The symmetry of L implies that $k \geq b/(2\epsilon)$ (else we could have used our space more efficiently by swapping b and $2\epsilon k$ ($b' = 2\epsilon k, k' = b/2\epsilon$) yielding the same values of $L(b', 2\epsilon k')$). $b'k' = bk$ implying that our space requirements were equivalent, but the larger value of k' would mean that we cover more observations (as long as $\epsilon < 0.5$). Consequently, we would never choose $k < b/(2\epsilon)$ if we were trying to minimize space.)

On the other hand, if we choose k too large relative to b , we would again use space inefficiently. Assume, for contradiction, that a space-efficient b, k existed, such that $k > 15b/\epsilon$. But, if so, we could more efficiently choose $k' = k/2$ and $b' = 2b$, using the same space but covering more observations. b' and k' would cover more observations because $k/2 \binom{2b+\epsilon k}{\epsilon k} > k \binom{b+2\epsilon k}{2\epsilon k}$, when $k > 15b/\epsilon$. It follows that if a pair b, k is space-efficient, then k is bounded by $b/(2\epsilon) \leq k \leq 15b/\epsilon$.

If $k = O(b/\epsilon)$, then we need only find the minimum b such that $\frac{b}{2\epsilon} \binom{2b-2}{b-1} \approx n$. Roughly, $b = O(\log(2\epsilon n))$, and $k = O(\frac{1}{2\epsilon} \log(2\epsilon n))$, so $bk = O((1/(2\epsilon)) \times \log^2(2\epsilon n))$.

3.3 The GK Algorithm

The new MRL algorithm was designed to be the best possible algorithm within the MRL framework. Nevertheless, it still suffers from some inefficiencies. For a given memory requirement, bk , the precision is reduced (that is, ϵ is increased) by three factors. First, each time `COLLAPSE` is called, combining the α buffers together increases the gap between `rmin` and `rmax` of elements in that buffer by the sum of the gaps between the individual `rmin` and `rmax` in *all* the buffers—because algorithms do not maintain any information that can allow them to recover how the deleted entries in each buffer may have been interleaved. Second, each `COLLAPSE` invokes the `prune` operation, which increases ϵ by $1/(2k)$. Finally, as is true for all algorithms in the MRL framework, MRL keeps no per-entry information about `rmin` and `rmax` for individual entries. Rather, we must assume that every entry in a buffer has the worst $\text{rmax} - \text{rmin}$.

We next describe an algorithm due to Greenwald and Khanna [10] that overcomes some of these drawbacks by not using `combine` and `prune`. It yields an ϵ -approximate quantile summary using only $O((\log \epsilon n)/\epsilon)$ space.

The GK Summary Data Structure

At any point in time n , GK maintains a summary data structure $Q_{\text{GK}}(n)$ that consists of an ordered sequence of tuples which correspond to a subset of the observations seen thus far. For each observation v in Q_{GK} , we maintain implicit bounds on the minimum and the maximum possible rank of the observation v among the first n observations. Let $\text{rmin}_{\text{GK}}(v)$ and $\text{rmax}_{\text{GK}}(v)$ denote respectively the lower and upper bounds on the rank of v among the observations seen so far. Specifically, Q_{GK} consists of tuples t_0, t_1, \dots, t_{s-1} where each tuple $t_i = (v_i, g_i, \Delta_i)$ consists of three components: (i) a value v_i that corresponds to one of the elements in the data sequence seen thus far, (ii) the value g_i equals $\text{rmin}_{\text{GK}}(v_i) - \text{rmin}_{\text{GK}}(v_{i-1})$ (for $i = 0$, $g_i = 0$), and (iii) Δ_i equals $\text{rmax}_{\text{GK}}(v_i) - \text{rmin}_{\text{GK}}(v_i)$. Note that $v_0 \leq v_1 \leq \dots \leq v_{s-1}$. We ensure that, at all times, the maximum and the minimum values are part of the summary. In other words, v_0 and v_{s-1} always correspond to the minimum and the maximum elements seen so far. It is easy to see that $\text{rmin}_{\text{GK}}(v_i) = \sum_{j \leq i} g_j$ and $\text{rmax}_{\text{GK}}(v_i) = \sum_{j \leq i} g_j + \Delta_i$. Thus $g_i + \Delta_i - 1$ is an upper bound on the *total* number of observations that may have fallen between v_{i-1} and v_i . Finally, observe that $\sum_i g_i$ equals n , the total number of observations seen so far.

Answering Quantile Queries

A summary of the above form can be used in a straightforward manner to provide ϵ -approximate answers to quantile queries. Proposition 1 forms the basis of our approach, and the following is an immediate corollary.

Corollary 3 *If at any time n , the summary $Q_{\text{GK}}(n)$ satisfies the property that $\max_i(g_i + \Delta_i) \leq 2\epsilon n$, then we can answer any ϕ -quantile query to within an ϵn precision.*

Overview

At a high level, our algorithm for maintaining the quantile summary proceeds as follows. Whenever the algorithm sees a new observation, it inserts in the summary a tuple corresponding to this observation. Periodically, the algorithm performs a sweep over the summary to “merge” some of the tuples into their neighbors so as to free up space. The heart of the algorithm is in this merge phase where we maintain several conditions that allow us to bound the space used by Q_{GK} at any time. We next develop some basic concepts that are needed to precisely describe these conditions.

Tuple Capacities

When a new tuple t_i is added to the summary at time n , we set its g_i value to be 1 and its Δ_i value² to be $\lfloor 2\epsilon n \rfloor - 1$. All summary operations maintain the property that the Δ_i value never changes. By Corollary 3, it suffices to ensure that at all times greater than $1/2\epsilon$, $\max_i(g_i + \Delta_i) \leq 2\epsilon n$. (For times earlier than $1/2\epsilon$ the summary preserves every observation, the error is always zero, and $\Delta_i = 0$ for all i .) Motivated by this consideration, we define the *capacity* of a tuple t_i at any time n' , denoted by $\text{cap}(t_i, n')$, to be $\lfloor 2\epsilon n' \rfloor - \Delta_i$. Thus the capacity of a tuple increases over time. An individual tuple is said to be *full* at time n' if $g_i + \Delta_i = \lfloor 2\epsilon n' \rfloor$. The *capacity* of an individual tuple is, therefore, the maximum number of observations that can be counted by g_i before the tuple becomes full.

Bands

Tuples with higher capacity correspond to values whose ranks are known with higher precision. Intuitively, high capacity tuples are more valuable than lower capacity tuples and our merge rules will favor elimination of lower capacity tuples by merging them into larger capacity ones. However, we will find it convenient to not differentiate among tuples whose capacities are within a small multiplicative factor of one another. We thus group tuples into geometric classes referred to as *bands* where, roughly speaking, a tuple t_i is in a band α if $\text{cap}(t_i, n) \approx 2^\alpha$. Since capacities increase over time, the band of a tuple increases over time. We will find it convenient to ensure the following stability property in assigning bands: if at some time n we have $\text{band}(t_i, n) = \text{band}(t_j, n)$, then for all times $n' \geq n$, we have $\text{band}(t_i, n') = \text{band}(t_j, n')$. We thus use a slightly more technical definition of bands. Let $p = \lfloor 2\epsilon n \rfloor$ and $\hat{\alpha} = \lceil \log_2 p \rceil$. Then we say $\text{band}(t_i, n)$ is α if

$$2^{\alpha-1} + (p \bmod 2^{\alpha-1}) \leq \text{cap}(t_i, n) < 2^\alpha + (p \bmod 2^\alpha).$$

If the Δ value of tuple t_i is p , then we say $\text{band}(t_i, n)$ is 0. It follows from the definition of band that at all times the first $1/(2\epsilon)$ observations, with $\Delta = 0$, are alone in $\text{band}_{\hat{\alpha}}$.

We will denote by $\text{band}(t_i, n)$ the band of tuple t_i at time n , and by $\text{band}_\alpha(n)$ all tuples (or equivalently, the capacities associated with these tuples) that have a band value of α . The terms $(p \bmod 2^{\alpha-1})$ and $(p \bmod 2^\alpha)$ above ensure the following stability property: once a pair of tuples is in the same band, they stay together from there on even as band boundaries are modified. At the same time, these terms do not change the underlying geometric grouping in any fundamental manner; $\text{band}_\alpha(n)$ contains either $2^{\alpha-1}$ or 2^α distinct capacity values.

²In practice, we set Δ more tightly by inserting $(v, 1, g_i + \Delta_i - 1)$ as the tuple immediately preceding t_{i+1} . It is easy to see that if Corollary 3 is satisfied before insertion, it remains true after insertion, and that $\lfloor 2\epsilon n \rfloor - 1$ is an upper bound on the value of Δ_i . For the purpose of our analysis, we always assume the worst-case insertion value of $\Delta_i = \lfloor 2\epsilon n \rfloor - 1$.

Proposition 3

1. At any point in time n and for any $\alpha, \hat{\alpha} > \alpha \geq 1$, the number of distinct capacity values that can belong to $\text{band}_\alpha(n)$ is either $2^{\alpha-1}$ or 2^α .
2. If at some time n , any two tuples t_i, t_j are in the same band, then for all times $n' \geq n$, this holds true.
3. At any point in time, n , and for any $\alpha, \hat{\alpha} \geq \alpha \geq 0$, the number of distinct capacity values that can belong to $\text{band}_\alpha(n)$ is $\leq 2^\alpha$.

Proof To see claim 1, if $p \bmod 2^\alpha < 2^{\alpha-1}$, then $p \bmod 2^\alpha = p \bmod 2^{\alpha-1}$, and $\text{band}_\alpha(n)$ contains $2^\alpha - 2^{\alpha-1} = 2^{\alpha-1}$ distinct values of capacity. If $p \bmod 2^\alpha \geq 2^{\alpha-1}$, then $p \bmod 2^\alpha = 2^{\alpha-1} + (p \bmod 2^{\alpha-1})$, and $\text{band}_\alpha(n)$ contains 2^α distinct capacity values.

To see claim 2, consider any $\text{band}_\alpha(n)$. Each time p increases by 1, if $p \bmod 2^\alpha \notin \{0, 2^{\alpha-1}\}$, then both $p \bmod 2^{\alpha-1}$ and $p \bmod 2^\alpha$ increase by 1 and thus the range of $\text{band}_\alpha(n)$ shifts by 1. At the same time, capacity of each tuple changes by 1 and thus the set of tuples that belong to $\text{band}_\alpha(n)$ stays unchanged.

Now suppose when p increases by 1, $p \bmod 2^\alpha = 2^{\alpha-1}$. It is easy to verify that for any value of p , there is at most one value of α that satisfies the equation $p \bmod 2^\alpha = 2^{\alpha-1}$. Then the left boundary of the band α decreases by $2^{\alpha-1} - 1$ while the right boundary increases by 1. The resulting band now captures all tuples that belonged to old bands $(\alpha - 1)$ and α . Also, for all bands β where $1 \leq \beta < \alpha$ (and hence $p \bmod 2^\beta = 0$), the range of the band changes from $[2^\beta - 1, 2^{\beta+1} - 1)$ to $[2^{\beta-1}, 2^\beta)$. Thus all tuples in the old band β now belong together to the new band $\beta + 1$. Finally, all bands $\gamma > \alpha$ satisfy $p \bmod 2^\gamma \notin \{0, 2^{\gamma-1}\}$, and hence continue to capture the same set of tuples as observed above.

Thus once a pair of tuples is present in the same band, their bands never diverge again.

Claim 3 holds for both band_0 and $\text{band}_{\hat{\alpha}}$ —they both contain precisely one distinct capacity value. For all other values of α , claim 3 follows directly from claim 1. \square

A Tree Representation

In order to decide on how tuples are merged in order to *compress* the summary, we create an *ordered* tree structure, referred to as the *quantile tree*, whose nodes correspond to the tuples in the summary. The tree structure creates a hierarchy based on tuple capacities and proximity. Specifically, the quantile tree $T(n)$ at time n is created from the summary $Q_{GK}(n) = \langle t_0, t_1, \dots, t_{s-1} \rangle$ as follows. $T(n)$ contains a node V_i for each t_i along with a special root node R . The parent of a node V_i is the node V_j such that j is the least index greater than i with $\text{band}(t_j, n) > \text{band}(t_i, n)$. If no such index exists, then the node R is set to be the parent. The children of each node are ordered as they appear in the summary. All children (and all descendants) of a given node V_i correspond to tuples that have capacities smaller than that of

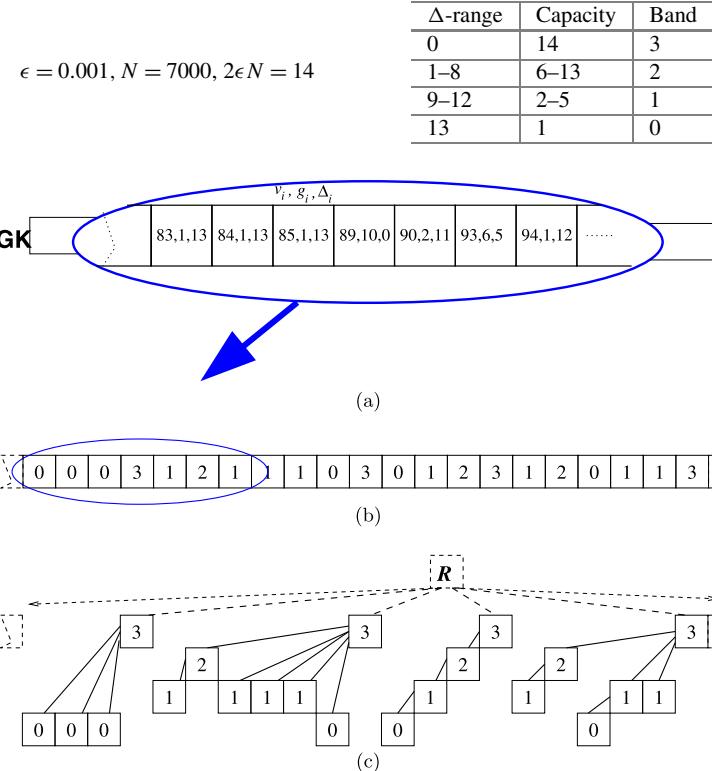


Fig. 4 (a) Tuple representation. (b) Tuples labeled only with band numbers. (c) Corresponding tree representation

tuple t_i . The relationship between $Q_{GK}(n)$ and $T(n)$ is represented pictorially in Fig. 4. We next highlight two useful properties of $T(n)$.

Proposition 4 *Each node V_i in a quantile tree $T(n)$ satisfies the following two properties:*

1. *The children of each node V_i in $T(n)$ are always arranged in non-increasing order of band in $Q_{GK}(n)$.*
2. *The tuples corresponding to V_i and the set of all descendants of V_i in $T(n)$ form a contiguous segment in $Q_{GK}(n)$.*

Proof To see property 1, consider any two children V_j and $V_{j'}$ of V_i with $j < j'$. Then if $\text{band}(t_{j'}, n) > \text{band}(t_j, n)$, $V_{j'}$ and not V_i would be the parent of node V_j in $T(n)$.

We establish property 2 by contradiction. Consider a node V_i that violates the property. Let k be the largest integer such that V_k is a descendant of V_i , and let j be the largest index less than k such that V_j is a descendant of V_i while V_{j+1} is

not. Also, let $j < \ell < k$ be the largest integer such that V_ℓ is not a descendant of V_i . Clearly, V_i must be the parent of V_k , and some node V_x where $x > \ell$ must be the parent of V_j .

If $\text{band}(t_\ell, n) < \text{band}(t_k, n)$, then one of the nodes $V_{\ell+1}, \dots, V_k$ must be the parent of V_ℓ —a contradiction since each of these nodes is a descendant of V_i by our choice of ℓ . Otherwise, we have $\text{band}(t_\ell, n) \geq \text{band}(t_k, n)$. Now if $\text{band}(t_\ell, n) \geq \text{band}(t_i, n)$, then V_j 's parent is some node $V_{x'}$ with $x' \leq \ell$. So it must be that $\text{band}(t_k, n) \leq \text{band}(t_\ell, n) < \text{band}(t_i, n)$. Consider the node V_y that is the parent of V_ℓ in $T(n)$. By our choice of V_ℓ , we have $k < y < i$. Since $\text{band}(V_y, n) > \text{band}(V_\ell, n)$, it must be that V_y is the parent of V_k and not V_i . A contradiction! \square

Operations

We now describe the various operations that we perform on our summary data structure. We start with a description of external operations:

External Operations

QUANTILE(ϕ) To compute an ϵ -approximate ϕ -quantile from the summary $Q_{GK}(n)$ after n observations, compute the rank, $r = \lceil \phi(n - 1) \rceil$. Find i such that both $r - \text{rmin}_{GK}(v_i) \leq \epsilon n$ and $\text{rmax}_{GK}(v_i) - r \leq \epsilon n$ and return v_i .

INSERT(v) Find the smallest i , such that $v_{i-1} \leq v < v_i$, and insert the tuple $(v, 1, \lfloor 2\epsilon n \rfloor - 1)$, between t_{i-1} and t_i . Increment s . As a special case, if v is the new minimum or the maximum observation seen, then insert $(v, 1, 0)$.

INSERT(v) maintains correct relationships between g_i , Δ_i , $\text{rmin}_{GK}(v_i)$ and $\text{rmax}_{GK}(v_i)$. Consider that if v is inserted before v_i , the value of $\text{rmin}_{GK}(v)$ may be as small as $\text{rmin}_{GK}(v_{i-1}) + 1$, and hence $g_i = 1$. Similarly, $\text{rmax}_{GK}(v)$ may be as large as the *current* $\text{rmax}_{GK}(v_i)$, which in turn is bounded to be within $\lfloor 2\epsilon n \rfloor$ of $\text{rmin}_{GK}(v_{i-1})$. Note that $\text{rmin}_{GK}(v_i)$ and $\text{rmax}_{GK}(v_i)$ increase by 1 after insertion.

Internal Operations

COMPRESS() The operation COMPRESS repeatedly attempts to find a suitable segment of adjacent tuples in the summary $Q_{GK}(n)$ and merges them into the neighboring tuple (i.e., the tuple that succeeds them in the summary). We first describe how the summary is updated when for any $1 < x \leq y$, tuples t_x, \dots, t_y are merged together into the neighboring tuple t_{y+1} . We replace g_{y+1} by $\sum_{j=x}^{y+1} g_j$ and Δ_{y+1} remains unchanged. It is easy to verify that this operation correctly maintains $\text{rmin}_{GK}(t_{y+1})$ and $\text{rmax}_{GK}(t_{y+1})$ values for all the tuples in the summary. Deletion of t_x, \dots, t_y does not alter the $\text{rmin}_{GK}()$ and $\text{rmax}_{GK}()$ values for any of the remaining tuples and the merge operation above precisely maintains this property.

```

COMPRESS()
for i from s - 2 to 0 do
    if ((band( $t_i$ , n)  $\leq$  band( $t_{i+1}$ , n)) &&
        ( $g_i^* + g_{i+1} + \Delta_{i+1} < 2\epsilon n$ )) then
         $g_{i+1} = g_{i+1} + g_i^*$ ;
        Remove  $t_i$  and all its descendants;
    end if
end for
end COMPRESS

```

Fig. 5 Pseudocode for COMPRESS

Initial State
$Q_{GK} \leftarrow \emptyset; s = 0; n = 0.$
Algorithm
<i>To add the $(n + 1)^{st}$ observation, v, to summary $Q_{GK}(n)$:</i>
if $(n \equiv 0 \bmod \frac{1}{2\epsilon})$ then
COMPRESS();
end if
INSERT(v);
$n = n + 1;$

Fig. 6 Pseudocode for the algorithm

COMPRESS chooses the segments to be merged in a specific manner, namely, it considers only those segments that correspond to a tuple t_i and all its descendants in the tree $T(n)$. By Lemma 4, we know that t_i and all its descendants corresponds to a segment of adjacent tuples $t_{i-a}, t_{i-a+1}, \dots, t_i$ in $Q_{GK}(n)$. Let g_i^* denote the sum of g -values of the tuple t_i and *all its descendants* in $T(n)$, that is, $g_i^* = \sum_{j=i-a}^i g_j$. To maintain the ϵ -approximate guarantee for the summary, we ensure that a merge is done only if $g_i^* + g_{i+1} + \Delta_{i+1} \leq 2\epsilon n$. Finally, COMPRESS ensures that we always merge into tuples of comparable or better capacity. The operation COMPRESS terminates if and only if there are no segments that satisfy the conditions above. Figure 5 describes an efficient implementation of the COMPRESS operation, and Fig. 6 depicts the pseudocode for the overall algorithm.

Note that since COMPRESS never alters the Δ value of surviving tuples, it follows that Δ_i of any quantile entry remains unchanged once it has been inserted.

COMPRESS inspects tuples from right (highest index) to left. Therefore, it first combines children (and their entire subtree of descendants) into parents. It combines siblings only when no more children can be combined into the parent.

Analysis

The INSERT as well as COMPRESS operations always ensure that $g_i + \Delta_i \leq 2\epsilon n$. As n always increases, it is easy to see that the data structure above maintains an ϵ -approximate quantile summary at each point in time. We will now establish that

the total number of tuples in the summary Q_{GK} after n observations have been seen is bounded by $(11/2\epsilon) \log(2\epsilon n)$.

We start by defining a notion of coverage. We say that a tuple t in the quantile summary Q_{GK} *covers* an observation v at any time n if either the tuple for v has been directly merged into t_i or a tuple t that covered v has been merged into t_i . Moreover, a tuple always covers itself. It is easy to see that the total number of observations covered by t_i is exactly given by $g_i = g_i(n)$. The lemmas below highlight some useful properties concerning coverage of observations by various tuples.

Lemma 5 *At no point in time, a tuple t with a band value of α covers a tuple t' which if it were alive, would have a band value strictly greater than α .*

Proof Note that the band of a tuple at any time n is completely determined by its Δ value. Since the Δ value never changes once a tuple is created, the notion of band value of a tuple is well-defined even if the tuple no longer exists in the summary.

Now suppose at some time n , the event described in the lemma occurs. The COMPRESS subroutine never merges a tuple t_i into an adjacent tuple t_{i+1} if the band of t_i is greater than the band of t_{i+1} . Thus the only way in which this event can occur is if it at some earlier time $m < n$, we had $\text{band}(t_i, m) \leq \text{band}(t_{i+1}, m)$, and at the current time n , we have $\text{band}(t_i, n) > \text{band}(t_{i+1}, n)$. Consider first the case when $\text{band}(t_i, m) = \text{band}(t_{i+1}, m)$. By Proposition 3, it cannot be the case that at some later time n , $\text{band}(t_i, n) \neq \text{band}(t_{i+1}, n)$. Now consider the case when $\text{band}(t_i, m) < \text{band}(t_{i+1}, m)$. Then $\Delta_i > \Delta_{i+1}$, and hence $\text{band}(t_i, n) \leq \text{band}(t_{i+1}, n)$ for all n . \square

Lemma 6 *At any point in time n , and for any integer α , the total number of observations covered cumulatively by all tuples with band values in $[0..\alpha]$ is bounded by $2^\alpha/\epsilon$.*

Proof By Proposition 3, each $\text{band}_\beta(n)$ contains at most 2^β distinct values of Δ . There are no more than $1/2\epsilon$ observations with any given Δ , so at most $2^\beta/2\epsilon$ observations were inserted with $\Delta \in \text{band}_\beta$. By Lemma 5, no observations from bands $> \alpha$ will be covered by a node from α . Therefore, the nodes in question can cover, at most, the total number of observations from all bands $\leq \alpha$. Summing over all $\beta \leq \alpha$ yields an upper bound of $2^{\alpha+1}/2\epsilon = 2^\alpha/\epsilon$. \square

The next lemma shows that for any given band value α , only a small number of nodes can have a child with that band value.

Lemma 7 *At any time n and for any given α , there are at most $3/2\epsilon$ nodes in $T(n)$ that have a child with band value of α . In other words, there are at most $3/2\epsilon$ parents of nodes from $\text{band}_\alpha(n)$.*

Proof Let m_{\min} and m_{\max} respectively denote the earliest and the latest times at which an observation in $\text{band}_\alpha(n)$ could be seen. It is easy to verify that

$$m_{\min} = \frac{2\epsilon n - 2^\alpha - (2\epsilon n \bmod 2^\alpha)}{2\epsilon} \quad \text{and}$$

$$m_{\max} = \frac{2\epsilon n - 2^{\alpha-1} - (2\epsilon n \bmod 2^{\alpha-1})}{2\epsilon}.$$

Thus, any parent of a node in $\text{band}_\alpha(n)$ must have $\Delta_i < 2\epsilon m_{\min}$.

Fix a parent node V_i with at least one child in $\text{band}_\alpha(n)$ and let V_j be the right-most such child. Denote by m_j the time at which the observation corresponding to V_j was seen.

We will show that at least a $(2\epsilon/3)$ -fraction of all observations that arrived after time m_{\min} can be uniquely mapped to the pair (V_i, V_j) . This in turn implies that no more than $3/2\epsilon$ such V_i 's can exist, thus establishing the lemma. The main idea underlying our proof is that the fact that COMPRESS did not merge V_j into V_i implies there must be a large number of observations that can be associated with the parent-child pair (V_i, V_j) .

We first claim that $g_j^*(n) + \sum_{k=j+1}^{i-1} g_k(n) \geq g_{i-1}^*(n)$. If $j = i - 1$, it is trivially true. Otherwise, the tuple t_{i-1} is distinct from t_j , and since V_j is a child of V_i (and not V_{i-1}), we know that $\text{band}(t_{i-1}, n) \leq \text{band}(t_j, n)$. Thus no tuple among t_1, t_2, \dots, t_j could be a descendant of t_{i-1} . Therefore, $\sum_{k=j+1}^{i-1} g_k(n) \geq g_{i-1}^*(n)$ and the claim follows.

Now since COMPRESS did not merge V_j into V_i , it must be the case that $g_{i-1}^*(n) + g_i(n) + \Delta_i > 2\epsilon n$. Using the claim above, we can conclude that $g_j^*(n) + \sum_{k=j+1}^{i-1} g_k(n) + g_i(n) + \Delta_i > 2\epsilon n$. Also, at time m_j , we had $g_i(m_j) + \Delta_i < 2\epsilon m_j$. Since m_j is at most m_{\max} , it must be that

$$g_j^*(n) + \sum_{k=j+1}^{i-1} g_k(n) + (g_i(n) - g_i(m_j)) > 2\epsilon(n - m_{\max}).$$

Finally, observe that for any other such parent–child pair, $V_{i'}$ and $V_{j'}$, the observations counted above by (V_j, V_i) and $(V_{j'}, V_{i'})$ are distinct. Since there are at most $n - m_{\min}$ total observations that arrived after m_{\min} , we can bound the total number of such pairs by

$$\frac{n - m_{\min}}{2\epsilon(n - m_{\max})}$$

which can be verified to be at most $3/2\epsilon$. □

We say that adjacent tuples (t_{i-1}, t_i) constitute a *full pair of tuples* at time n' , if $g_{i-1} + g_i + \Delta_i > \lfloor 2\epsilon n' \rfloor$. Given such a full pair of tuples, we say that the tuple t_{i-1} is a *left partner* and t_i is a *right partner* in this full pair.

Lemma 8 *At any time n and for any given α , there are at most $4/\epsilon$ tuples from $\text{band}_\alpha(n)$ that are right partners in a full pair of tuples.*

Proof Let X be the set of tuples in $\text{band}_\alpha(n)$ that participate as a right partner in some full pair. We first consider the case when tuples in X form a single contiguous segment in $\text{Q}_{\text{GK}}(n)$. Let t_i, \dots, t_{i+p-1} be a maximal contiguous segment of

$\text{band}_\alpha(n)$ tuples in $\mathbf{Q}_{\text{GK}}(n)$. Since these tuples are alive in $\mathbf{Q}_{\text{GK}}(n)$, it must be the case that

$$g_{j-1}^* + g_j + \Delta_j > 2\epsilon n \quad i \leq j < i+p.$$

Adding over all j , we get

$$\sum_{j=i}^{i+p-1} g_{j-1}^* + \sum_{j=i}^{i+p-1} g_j + \sum_{j=i}^{i+p-1} \Delta_j > 2p\epsilon n.$$

In particular, we can conclude that

$$2 \sum_{j=i-1}^{i+p-1} g_j^* + \sum_{j=i}^{i+p-1} \Delta_j > 2p\epsilon n.$$

The first term in the LHS of the above inequality counts twice the number of observations covered by nodes in $\text{band}_\alpha(n)$ or by one of its descendants in the tree $T(n)$. Using Lemma 6, this sum can be bounded by $2(2^\alpha/\epsilon)$. The second term can be bounded by $p(2\epsilon n - 2^{\alpha-1})$ since the largest possible Δ value for a tuple with a band value of α or less is $(2\epsilon n - 2^{\alpha-1})$. Substituting these bounds, we get

$$\frac{2^{\alpha+1}}{\epsilon} + p(2\epsilon n - 2^{\alpha-1}) > 2p\epsilon n.$$

Simplifying above, we get $p < 4/\epsilon$ as claimed by the lemma. Finally, the same argument applies when nodes in X induce multiple segments in $\mathbf{Q}_{\text{GK}}(n)$; we simply consider the above summation over all such segments. \square

Lemma 9 *At any time n and for any given α , the maximum number of tuples possible from each $\text{band}_\alpha(n)$ is $11/2\epsilon$.*

Proof By Lemma 8, we know that the number of $\text{band}_\alpha(n)$ nodes that are right partners in some full pair can be bounded by $4/\epsilon$. Any other $\text{band}_\alpha(n)$ node either does not participate in any full pair or occurs only as a left partner. We first claim that each parent of a $\text{band}_\alpha(n)$ node can have at most one such node in $\text{band}_\alpha(n)$. To see this, observe that if a pair of non-full adjacent tuples t_i, t_{i+1} , where $t_{i+1} \in \text{band}_\alpha(n)$, is not merged then it must be because $\text{band}(t_i, n)$ is greater than α . But Proposition 4 tells us that this event can occur only once for any α , and therefore, V_{i+1} must be the unique $\text{band}_\alpha(n)$ child of its parent that does not participate in a full pair. It is also easy to verify that for each parent node, at most one $\text{band}_\alpha(n)$ tuple can participate only as a left partner in a full pair. Finally, observe that only one of the above two events can occur for each parent node. By Lemma 7, there are at most $3/2\epsilon$ parents of such nodes, and thus the total number of $\text{band}_\alpha(n)$ nodes can be bounded by $11/2\epsilon$. \square

Theorem 2 *At any time n , the total number of tuples stored in $\mathbf{Q}_{\text{GK}}(n)$ is at most $(11/2\epsilon) \log(2\epsilon n)$.*

Proof There are at most $1 + \lfloor \log 2\epsilon n \rfloor$ bands at time n . There can be at most $3/2\epsilon$ total tuples in $Q_{\text{GK}}(n)$ from bands 0 and 1. For the remaining bands, Lemma 9 bounds the maximum number of tuples in each band. The result follows. \square

4 Randomized Algorithms

We present here two distinct approaches for using randomization to reduce the space needed. The first approach essentially samples the input elements, and presents the sample as input to a deterministic algorithm. The second approach uses hashing to randomly cluster the input elements, thus reducing the number of distinct input elements seen by the quantile summary. The sampling-based approaches presented here work only for the cash-register model. The hashing-based approach, on the other hand, works in the more general turnstile model that allows for deletion of elements. However, this latter approach requires that we know the number of elements in the input sequence in advance.

4.1 Sampling-Based Approaches

Sampling of elements offers a simple and effective way to reduce the space needed to create quantile summaries. In particular, the space needed can be made independent of the size of the data stream, if we are willing to settle for a probabilistic guarantee on the precision of the summary generated. The idea is to draw a random sample from the input, and run a deterministic algorithm on the sample to generate the quantile summary. The size of the sample depends only on the probabilistic guarantee and the desired precision for the summary. The following lemma from Manku et al. [14] serves as a basis for this approach.

Lemma 10 ([14]) *Let $\epsilon, \delta \in (0, 1)$, and let S be a set of n elements. There exists an integer $p = \Theta(\frac{1}{\epsilon^2} \log(\frac{1}{\epsilon\delta}))$ such that if we sample p elements from S uniformly at random, and create an $\epsilon/2$ -approximate quantile summary Q on the sample, then Q is an ϵ -approximate summary for S with probability at least $1 - \delta$.*

If the length of the input sequence is known in advance, we can easily draw a sample of size p as required above, and maintain an $\epsilon/2$ -approximate quantile summary Q on the sample. Total space used by this approach is $O(\frac{1}{\epsilon} \log(\epsilon p))$, giving us the following theorem.

Theorem 3 *For any $\epsilon, \delta \in (0, 1)$, we can compute with probability at least $1 - \delta$, an ϵ -approximate quantile summary for a sequence of n elements using $O(\frac{1}{\epsilon} \log(\frac{1}{\epsilon}) + \frac{1}{\epsilon} \log \log(\frac{1}{\epsilon\delta}))$ space, assuming the sequence size n is known in advance.*

When the length of the input sequence is not known a priori, one approach is to use a technique called *reservoir sampling* (discussed in detail in another chapter in this handbook) that maintains a uniform sample at all times. However, the elements in the sample are constantly being replaced as the length of the input sequence increases, and thus the quantile summary cannot be constructed incrementally. The sample must be stored explicitly and the observations can be fed to the deterministic algorithm only when we stop, and are certain the elements in the sample will not be replaced. Since the sample size dominates the space needed in this case, we get the following theorem.

Theorem 4 *For any $\epsilon, \delta \in (0, 1)$, we can compute with probability at least $1 - \delta$, an ϵ -approximate quantile summary for a sequence of n elements using $O(\frac{1}{\epsilon^2} \log(\frac{1}{\epsilon\delta}))$ space.*

Manku et al. [15] used a non-uniform sampling approach to get around the large space requirements imposed by the reservoir sampling. We state their main result below and refer the reader to the paper for more details.

Theorem 5 ([15]) *For any $\epsilon, \delta \in (0, 1)$, we can compute with probability at least $1 - \delta$, an ϵ -approximate quantile summary for a sequence of n elements using $O(\frac{1}{\epsilon} \log^2(\frac{1}{\epsilon}) + \frac{1}{\epsilon} \log^2 \log(\frac{1}{\epsilon\delta}))$ space.*

4.2 The Count-Min Algorithm

The Count-Min (CM) algorithm [4] is a randomized approach for maintaining quantiles when the universe size is known in advance. Suppose all elements are drawn from a universe $U = \{1, 2, \dots, M\}$ of size M . The CM algorithm uses $O(\frac{1}{\epsilon} (\log^2 M) (\log(\frac{\log M}{\epsilon\delta})))$ space to answer any quantile query with ϵ -accuracy with probability at least $1 - \delta$. This is in contrast to the $O(\frac{1}{\epsilon} \log(\epsilon n))$ space used by the deterministic GK algorithm. The two space bounds are incomparable in the sense that their relative quality depends on the relation between n and M . In addition to being quite simple, the main strength of the CM algorithm is that it works in the more general turnstile model, provided all element counts are non-negative throughout its execution. We start with a description of the basic data structure maintained by the CM algorithm and then describe how the data structure is adapted to handle quantile queries. A key concept underlying the CM data structure is that of universal hash families.

Universal Hash Families

For any positive integer m , a family $\mathcal{H} = \{h_1, \dots, h_k\}$ of hash functions where each $h_i : U \rightarrow [1..m]$ is a *universal hash family* if for any two distinct elements $x, y \in U$,

we have

$$\Pr_{h_i \in \mathcal{H}}[h_i(x) = h_i(y)] \leq 1/m.$$

The set of all possible hash functions $h : U \rightarrow [1..m]$ is easily seen to be a hash family, but the number of hash functions in this family is exponentially large. A beautiful result of Carter and Wegman [3] shows that there exist universal hash families with only $O(M^2)$ hash functions that can be constructed in polynomial-time. Moreover, any function in the family can be described completely using $O(\log M)$ bits.

We are now ready to describe the basic CM data structure.

Basic CM Data Structure

An (ϵ_0, δ_0) CM data structure consists of a $p \times q$ table T where $p = \lceil \ln(1/\delta_0) \rceil$ and $q = \lceil e/\epsilon_0 \rceil$, and a universal hash family \mathcal{H} such that each $h \in \mathcal{H}$ is a function $h : U \rightarrow [1..q]$. We associate with each row $i \in [1..p]$ a hash function h_i chosen uniformly at random from the hash family \mathcal{H} . The table is initialized to all zeroes at the beginning. Whenever an update (x, c_x) arrives for some $x \in U$, we modify for each $1 \leq i \leq p$:

$$T[i, h_i(x)] = T[i, h_i(x)] + c_x.$$

At any point in time t , let $\mathbf{C}(t) = (C_1, \dots, C_M)$ where C_x denotes the sum $\{\sum c_y \mid (y, c_y) \text{ arrived before time } t\}$. When the time t is clear from context, we will simply use \mathbf{C} .

Given a query for C_x , the CM data structure outputs the estimate $\hat{C}_x = \min_{1 \leq i \leq p} T[i, h_i(x)]$. The lemma below gives useful properties of the estimate $\hat{C}(x)$.

Lemma 11 *Let $x \in U$ be any fixed element. Then at any time t , with probability at least $1 - \delta_0$:*

$$C_x \leq \hat{C}_x \leq C_x + \epsilon_0 \|\mathbf{C}\|_1.$$

Proof Recall that by assumption, $C_y \geq 0$ for all $y \in U$ at all times t . It is then easy to see that $\hat{C}_x \geq C_x$ at all times since each update (x, c_x) leads to increment of $T[i, h_i(x)]$ by c_x for each $1 \leq i \leq p$. In addition, any element y such that $h_i(y) = h_i(x)$ may contribute to $T[i, h_i(x)]$ as well but this contribution is guaranteed to be non-negative by our assumption.

We now bound the probability that $\hat{C}_x > C_x + \epsilon_0 \|\mathbf{C}\|_1$ at any time t . Fix an element $x \in U$ and an $i \in [1..p]$. We start by analyzing the probability of the event that $T[i, h_i(x)] > C_x + \epsilon_0 \|\mathbf{C}\|_1$. Let $Z_i(x)$ be a random variable that is defined to be $|\{\sum_{y \in U \setminus \{x\}} C_y \mid h_i(y) = h_i(x)\}|$. Since h_i is drawn uniformly at random from a universal hash family, we have

$$E[Z_i(x)] = \sum_{y \in U} \Pr[h_i(y) = h_i(x)] C_y \leq \frac{\sum_{y \in U \setminus \{x\}} C_y}{q} \leq \frac{\epsilon_0 \|\mathbf{C}\|_1}{e}.$$

Then by Markov's inequality, we have that

$$\Pr[T[i, h_i(x)] > C_x + \epsilon_0 \|\mathbf{C}\|_1] \leq \Pr[T[i, h_i(x)] > \epsilon_0 \|\mathbf{C}\|_1] \leq \frac{1}{e}.$$

Thus

$$\Pr\left[\min_{1 \leq i \leq p} T[i, h_i(x)]\right] > C_x + \epsilon_0 \|\mathbf{C}\|_1 \leq \left(\frac{1}{e}\right)^{\ln(1/\delta_0)} \leq \delta_0. \quad \square$$

CM Data Structure for Quantile Queries

In order to support quantile queries, we need to modify the basic CM data structure to support range queries. A range query $R[\ell, r]$ specifies two elements $\ell, r \in U$ and asks for $\sum_{\ell \leq x \leq r} C_x$. Suppose we are given a data structure that can answer with probability at least $1 - \delta$ every range query to within an additive error of at most $\epsilon \|\mathbf{C}\|_1$. Then this data structure can be used to answer any ϕ -quantile query with ϵ -accuracy with probability at least $1 - \delta$. The idea is to perform a binary search for the smallest element $r \in U$ such that $\hat{R}[1, r] \geq \phi \|\mathbf{C}\|_1$. We output the element r as the ϕ -quantile. Clearly, it is an ϵ -accurate ϕ -quantile with probability at least $1 - \delta$.

We now describe how the basic CM data structure can be modified to support range queries. For clarity of exposition, we will assume without any loss of generality that $M = 2^u$ for some integer u . We will define a collection of CM data structures, say, $\text{CM}_0, \text{CM}_1, \dots, \text{CM}_u$ such that CM_i can answer any range query of the form $R[j2^i + 1, (j+1)2^i]$ with an additive error of at most $\frac{\epsilon \|\mathbf{C}\|_1}{(u+1)}$. Then to answer a range query $R[1, r]$ for any $r \in [1..M]$, we consider the binary representation of r . Let $i_1 > i_2 > \dots > i_b$ denote the bit positions with a 1 in the representation. In response to the query $R[1, r]$, we return

$$\begin{aligned} \hat{R}[1, r] = & \hat{R}[1, 2^{i_1}] + \hat{R}[2^{i_1} + 1, 2^{i_1} + 2^{i_2}] + \dots \\ & + \hat{R}[2^{i_1} + \dots + 2^{i_{b-1}} + 1, 2^{i_1} + \dots + 2^{i_{b-1}} + 2^{i_b}] \end{aligned}$$

where $\hat{R}[2^{i_1} + \dots + 2^{i_{j-1}} + 1, 2^{i_1} + \dots + 2^{i_{j-1}} + 2^{i_j}]$ is the value returned by CM_{i_j} in response to the query $R[2^{i_1} + \dots + 2^{i_{j-1}} + 1, 2^{i_1} + \dots + 2^{i_{j-1}} + 2^{i_j}]$. Since each term in the RHS has an additive error of at most $\frac{\epsilon \|\mathbf{C}\|_1}{(u+1)}$, we know that

$$R[1, r] \leq \hat{R}[1, r] \leq R[1, r] + \epsilon \|\mathbf{C}\|_1.$$

The Data Structure CM_i

It now remains to describe the data structure CM_i for $0 \leq i \leq u$. Fix an $i \in [0..u]$, and let $u_i = u - i$. Define $U_i = \{x_{i,1}, \dots, x_{i,2^{u_i}}\}$ to be the universe underlying the data structure CM_i . The element $x_{i,j} \in U_i$ serves as the *unique representative* for

all elements in U that lie in the range $[j2^i + 1, (j + 1)2^i]$. Thus each element in U is covered by a unique element in U_i . The data structure CM_i is an (ϵ_0, δ_0) CM data structure over U_i where $\epsilon_0 = \frac{\epsilon}{(u+1)}$ and $\delta_0 = \frac{\delta}{(u+1)}$. Whenever an update (x, c_x) arrives for $x \in U_i$, we simply add c_x to the unique representative $x_{i,j} \in U_i$ that covers x .

The following is an immediate corollary of Lemma 11.

Corollary 4 *Let $j \in [1..2^{u-i}]$ be a fixed integer. The data structure CM_i can be used to answer the query $R[j2^i + 1, (j + 1)2^i]$ within an additive error of $\epsilon_0\|\mathbf{C}\|_1$ with probability at least $1 - \delta_0$.*

To answer a range query $R[1..r]$, we aggregate answers from up to $(u + 1)$ queries (to the data structures $\text{CM}_0, \text{CM}_1, \dots, \text{CM}_u$), each with an additive error of $\epsilon_0\|\mathbf{C}\|_1$ with probability at least $1 - \delta_0$. Using union bounds, we can thus conclude that with probability at least $1 - (u + 1)\delta_0 = 1 - \delta$, the total error is bounded by $(u + 1)\epsilon_0\|\mathbf{C}\|_1 = \epsilon\|\mathbf{C}\|_1$.

Application to Quantile Queries

In order to answer every ϕ -quantile query to within ϵ -accuracy, it suffices to be able to answer ϕ -quantile queries for ϕ -values restricted to be in the set $\{\epsilon/2, \epsilon, 3\epsilon/2, \dots\}$ with $\epsilon/2$ -accuracy. Given any arbitrary ϕ -quantile query, we can answer it by querying for a ϕ' -quantile and returning the answer, where

$$\phi' = \left\lceil \frac{\phi}{(\epsilon/2)} \right\rceil (\epsilon/2).$$

It is easy to see that any $(\epsilon/2)$ -accurate answer to the ϕ' -quantile is an ϵ -accurate answer to the ϕ -quantile query.

In order to answer every quantile query to within an additive error of $\epsilon\|\mathbf{C}\|_1$ with probability at least $1 - \delta$, each CM_i data structure is created with suitably chosen parameters ϵ_0 and δ_0 . Since any single range query requires aggregating together at most $(u + 1)$ answers, and there are $2/\epsilon$ quantile queries overall, it suffices to set

$$\epsilon_0 = \frac{\epsilon}{(u + 1)} \quad \text{and} \quad \delta_0 = \frac{\delta}{(u + 1)} \cdot \frac{2}{\epsilon}.$$

The space used by each CM_i data structure for this choice of parameters is $O(\frac{1}{\epsilon}(\log M)(\log(\frac{u}{\epsilon\delta})))$. Hence the overall space used by this approach is $O(\frac{1}{\epsilon}(\log^2 M)(\log(\frac{\log M}{\epsilon\delta})))$.

The CM algorithm strongly utilizes the knowledge of the universe size. In absence of deletions, stronger space bounds can be obtained by exploiting the knowledge of the universe size. For instance, the q -digest summary of Shrivastava et al. [18] described in Sect. 5.3 is an ϵ -approximate quantile summary that uses only $O(\frac{1}{\epsilon} \log M)$ space. Moreover, the precision guarantee of a q -digest is deterministic.

However, the strength of the CM algorithm is in its ability to handle deletions. To see another interesting example of an algorithm that uses randomization to handle deletions, the reader is referred to the RSS algorithm [8, 9].

5 Other Models

So far we have considered deterministic algorithms with absolute guarantees and randomized algorithms with probabilistic guarantees mainly in the setting of the *cash register* model [7]. However, quantile computations can be considered under different streaming models. We have seen in Sect. 4.2 that the cash register model can be extended to the *turnstile* model [17], in which the stream can include both insertions *and* deletions of observations. We can also consider settings in which the complete dataset is accessible, at a cost, allowing us to perform multiple (expensive) passes. Settings in which other features of this model have been varied have been studied as well. For example, one may know the types of queries in advance [15], or exploit prior knowledge of the precision and range of the data values [4, 8, 18].

While algorithms from two different settings cannot be directly compared, it is still worth understanding how they may be related. In this section, we will briefly consider a small sample of alternative models where the ideas presented in this chapter are directly applicable—either used as black box components in other algorithms, or adapted to a new setting with relatively minor modifications—and compare the modified algorithms to other algorithms in the literature. In each setting, we first briefly present an algorithm that follows naturally from the ideas presented in this chapter, then present an algorithm from the literature specifically designed for the new setting. Some of these models will be covered in more depth in later chapters in this book.

5.1 Deletions

In many database applications, a summary is stored with large data sets. For queries in which an approximate answer is sufficient, the query can be cheaply executed over the summary rather than over the entire, large, dataset. In such cases, we need to *Maintain* the summary in the face of operations on the underlying data set. This setting differs from our earlier model in an important way: both insertion and deletion operations may be performed on the underlying set. We focus here on *well-formed inputs* where each deletion corresponds uniquely to an earlier insertion. When deletions are possible, the size of the dataset can grow and shrink. The parameter n can no longer denote both the number of observations that have been seen so far and the current time. In the turnstile model we will, instead, denote the current time by t and let $n = n(t)$ denote the current number of elements in the data set, and $m = m(t)$ will denote $\max_{1 \leq i \leq t} n(i)$.

The difficulty in guaranteeing precise responses to quantile queries in the turnstile model lies in recovering information once it has been discarded from the summary. In particular, when there are only insertions, the error allowed in the ranks of elements in the summary grows monotonically. But when deletions occur, we may need to greatly refine the rank information for existing elements in the summary. For instance, if we insert n elements in a set, then the allowed error in the rank of any observation is ϵn . But now if we delete all but $1/\epsilon$ of the observations, then the ϵ -approximate property now requires us to know the rank and value of each remaining element in the set exactly!

However, we can deal with similarly useful, but more tractable, problems by investigating slightly more relaxed settings. Gibbons, Matias, and Poosala [5, 6] were the first to present an algorithm to maintain a form of quantile summary in the face of deletions in the case where multiple passes over the data set are possible, although expensive. In situations where a second pass is impossible, we relax the requirement that we return a value v that is (currently) in S . In this latter setting, we can gain some further traction by weakening the guarantees we offer. Deterministic algorithms can temper their precision guarantees as a function of the input pattern (performing better on “easy” input patterns, and worse on “hard” patterns), and randomized algorithms can offer probabilistic, rather than absolute, guarantees.

Deterministic Algorithms with Input Dependent Guarantees

We first extend the GK [10] algorithm in a natural way to support a $\text{DELETE}(v)$ operation. We will see that for certain input sequences we can maintain a guarantee of ϵ -precision in our responses to quantile queries—even in the face of deletions.

$\text{DELETE}(v)$ Find the smallest i , such that $v_{i-1} \leq v < v_i$. (Note that i may be 1 if the minimum element was already deleted.) To delete v we must update $\text{rmax}_{\text{GK}}(v_j)$ and $\text{rmin}_{\text{GK}}(v_j)$ for all observations stored in the summary. For all $j > i$, $\text{rmin}_{\text{GK}}(v_j)$ and $\text{rmax}_{\text{GK}}(v_j)$ are reduced by 1. Further, we know that $\text{rmax}_{\text{GK}}(v_{j-1}) < \text{rmax}_{\text{GK}}(v_j)$. If our estimate of $\text{rmax}_{\text{GK}}((v_j))$ is reduced, such that $\text{rmax}_{\text{GK}}(v_{j-1}) = \text{rmax}_{\text{GK}}(v_j)$, then decrement $\text{rmax}_{\text{GK}}(v_{j-1})$ by 1. Deletion of an observation covered by v_i is implemented by simply decrementing g_i . This decrements all $\text{rmin}_{\text{GK}}(v_j)$ and $\text{rmax}_{\text{GK}}(v_j)$, for $j > i$, as required. The pseudocode in Fig. 7 that decrements Δ_{i-1} maintains the invariant that $\text{rmax}_{\text{GK}}(v_{j-1}) < \text{rmax}_{\text{GK}}(v_j)$. Finally, v_i is removed from the summary if $g_i = 0$ and i was not one of the extreme ranking elements ($i = 0$, or $i = s - 1$).

Because we do not delete v_i until *all* observations covered by the tuple are also deleted, it is no longer the case that all $v_i \in Q_{\text{GK}}$ are members of the underlying set.

At time t , a GK summary $Q_{\text{GK}}(\epsilon)$ will never delete a tuple if the resulting gap would exceed $2\epsilon n(t)$. By Proposition 1, the precision of the resulting summary is the maximum, over all i , of $(\text{rmax}_{Q_{\text{GK}}(\epsilon)}(v_{i+1}) - \text{rmin}_{Q_{\text{GK}}(\epsilon)}(v_i))/(2n(t))$. In the simple setting without deletions, the actual precision of $Q_{\text{GK}}(\epsilon)$ is always $\leq \epsilon$. Unfortunately, it is impossible to bound this precision in the face of arbitrary input in

```

DELETE(v)
 $g_i = g_i - 1$ 
if ( $g_i = 0$ ) and
    ( $(i \neq s - 1)$  or ( $i \neq 0$ )) then
        Remove  $t_i$  from Q
    end if
for  $j = (i - 1)$  to 0
    if ( $\Delta_j \geq (g_{j+1} + \Delta_{j+1})$ )
        then  $\Delta_j = \Delta_j - 1$ 
        else break
    end if
end for

```

Fig. 7 Pseudocode for DELETE

the setting of the turnstile model. In particular, after deletions, $Q_{\text{GK}}(\epsilon')$ may not have precision ϵ' in the turnstile model. However, in many cases, application-specific behavior can allow us to bound the maximum $\text{rmax}_{\text{GK}}(v_{i+1}) - \text{rmin}_{\text{GK}}(v_i)$ and hence construct summaries with guaranteed ϵ precision.

Example: Bounded Deletions

Perhaps the simplest example of application-specific behavior is when we know, in advance, some bound on the impact of deletions on the size of the data set. Let $\alpha < 1$ denote a known fixed lower bound, such that for any time t , $n(t) > \alpha m(t)$. If $Q_{\text{GK}}(\epsilon')$ is an ϵ' -approximate quantile summary, then COMPRESS will never delete a tuple if the resulting gap would exceed $2\epsilon' n(t)$. Given that at all time t , $n(t) \leq m(t)$, we know that for all tuples $\text{rmax}_{\text{GK}}(v_{i+1}) - \text{rmin}_{\text{GK}}(v_i) \leq 2\epsilon' m(t)$, and the precision at time t is then bounded above by $\epsilon' m(t)/n(t)$. $n(t) > \alpha m(t)$, and therefore the precision $\epsilon' m(t)/n(t) \leq \epsilon'/\alpha$. Consequently, if we choose $\epsilon' = \alpha\epsilon$, then a summary $Q_{\text{GK}}(\epsilon')$ has precision ϵ even after deletions. This loose specification is too broad to derive specific bounds on the size of our data structures. We proceed now to a concrete example in which analysis of the application-specific behavior allows us to demonstrate stronger limits on the size and precision of the resulting summary.

Example: Session Data

The AT&T network monitors the distribution of the duration of *active* calls over time [8] through the collection of Call Detail Records (CDRs). The start-time of each call is inserted into the quantile summary when the call begins. When the call ends it is no longer active and the start-time is deleted from the summary. At any time there is one observation in the set for each active call, and the duration of the call is simply the difference between the current time and the stored start time of

the call. (The median duration is the current time minus the start time of the median observation in the set.)

We first show that the size of the GK summary structure will be constant.

Session data arrives at the summary in order of start time. Consequently, new observations are monotonically increasing (although deletions may occur in arbitrary order). Each new observation is a new maximum, and we know its rank exactly.

Proposition 5 *If the input sequence is monotonically increasing, the data structure $Q_{GK}(\epsilon)$ uses only $O(1/\epsilon)$ space.*

Proof The exact rank of each newly arriving observation is always known. Hence, the Δ value of each tuple in the summary is always 0. It then follows that all tuples are siblings in the tree representation. Consequently, after we run COMPRESS, each tuple except the leftmost tuple is a right partner of a full tuple pair, and $g_{i-1} + g_i + \Delta_i > 2\epsilon n(t)$ in $Q_{GK}(\epsilon)$. Thus if there are $(k+1)$ tuples in $Q_{GK}(\epsilon)$, then summing over the k full tuple pairs we get $\sum_{i=1}^k (g_{i-1} + g_i + \Delta_i) \geq k(2\epsilon n(t))$. Since $\sum \Delta_i = 0$ and $\sum g_i \leq n(t)$, we get $2n(t) \geq k(2\epsilon n(t))$, and $k \leq \frac{1}{\epsilon}$. Since we run COMPRESS after $2/\epsilon$ new observations are added, $Q_{GK}(\epsilon)$ contains $O(1/\epsilon)$ tuples at all times t , regardless of the size of $n(t)$. \square

Proposition 5 tells us only about the size of Q_{GK} . By Proposition 1, Q_{GK} will have precision ϵ' at time t if the difference in rank between any two consecutive stored tuples, $r_{\max_{GK}(v_{i+1})} - r_{\min_{GK}(v_i)}$, is $\leq 2\epsilon' n(t)$. In our example, the expected difference in rank follows from the observation that phone calls (for example, the trace data from [8]) are typically modeled as having exponentially distributed lifetimes.

Proposition 6 *For sessions with exponentially distributed lifetimes and monotonically increasing arrivals, for a given ϵ , the expected difference in rank between consecutive tuples in $Q_{GK}(\epsilon)$ at time t is $\leq 2\epsilon E(n(t))$, where $E(n(t))$ is the expected number of elements in S at time t .*

Proof Fix any i . Let t' denote the most recent time at which COMPRESS deleted any tuples that lay between v_i and v_{i+1} . Let Δr denote the difference ($r_{\max_{GK}(v_{i+1})} - r_{\min_{GK}(v_i)}$) at time t' . Δr must have been $\leq 2\epsilon n(t')$. Let $d = d(t', t)$ denote the total number of deletions that occurred between times t' and t . Then $n(t) \geq n(t') - d$.

The exponential distribution is “memoryless”—all calls are equally likely to terminate within a given interval. Therefore, if the probability that a call terminates within the interval $[t', t]$ is p , then the expected value of the total number of deletions, d , during $[t', t]$ is $pn(t')$. Similarly, the expected value of the number of deletions falling between v_i and v_{i+1} during $[t', t]$ is $p\Delta r$. At time t the expected value of ($r_{\max_{GK}(v_{i+1})} - r_{\min_{GK}(v_i)}$) is $(1-p)\Delta r$. The expected value of $n(t)$ is $\geq (1-p)n(t')$. Given that $\Delta r \leq 2\epsilon n(t')$, we have $(1-p)\Delta r \leq 2\epsilon(1-p)n(t') \leq 2\epsilon n(t)$.

By Proposition 5, the size of $Q_{GK}(\epsilon)$ will be $O(1/\epsilon)$. \square

To be more concrete, in the CDR trace data described in [8], the RSS summary has $\epsilon = 0.1$ precision, and has a maximum memory footprint of 11K bytes. Assuming that we store 12 bytes per tuple, the GK algorithm, in the same memory footprint, should be able to store more than 900 tuples. We expect the typical gap between stored tuples to be commensurate with a precision of roughly 0.0011—about 100 times more accurate than the RSS summary. This demonstrates the potential payoff of domain-specific analysis, but it is important to recall that this analysis provides no guarantees in the general case.

Probabilistic Guarantees Across All Inputs

The GK algorithm above exploited structure that was specific to the given example. However, more adversarial cases are easy to imagine. An adversary, for example, can delete all observations except for those that lie between two consecutive tuples in the summary. In such cases either there can be no precision guarantee (we will not be able to return even a single observation from the data set) or else the original “summary” must store every observation—providing no reduction in space. Thus, when details of the application are not known, algorithms such as GK are unsatisfactory. Even when the expected behavior of an application is known, there may be a chance that the input sequence is unexpectedly adversarial.

Fortunately, there is a much better approach than aiming for absolute guarantees in the face of deletions. Randomized algorithms that summarize the number of observations within a range of values (c.f. RSS [8] or CM [4]) can give probabilistic guarantees on precision and upper bounds on space for *any* application, without requiring case by case analysis. The Count-Min algorithm is described in Sect. 4.2. Assuming that observations can take on any one of M values, then CM summarizes a sample in space $O(\frac{1}{\epsilon}(\log^2 M)(\log(\frac{\log M}{\epsilon\delta})))$. This summary is ϵ -approximate with probability at least $1 - \delta$.

5.2 Sliding Window Model

In some applications, we need to compute order statistics over the W most recent observations, rather than over the entire stream. When W is small enough to fit into memory, it suffices to store the last W observations in a circular buffer and compute the statistics exactly. However, when W is itself very large, then we must *summarize* a *sliding window* of the most recent W elements of the stream. A sliding window is a case where observations are being deleted in a systematic fashion based on the time of arrival. The chief difficulty in such sliding window quantile summaries compared to the summaries over entire streams is that, as the window slides, we need to remove observations from the summary—as in the turnstile model. However, unlike the strict turnstile model where the deletions are delivered to the summary from an external source (and, we presume, are properly paired to an undeleted input), we do

not have a complete ordered record of observations, and hence do not know what value needs to be deleted at any given time. On the other hand, deletions in the sliding window model are not arbitrarily distributed; they have a nice structure that can be exploited.

Sliding windows may be either *fixed* or *variable* size. In a fixed sliding window of size W , each newly arrived observation (after the first W arrivals) is paired with a deletion of the oldest observation in the window. Thus, in the steady state, the window always covers precisely W elements. Variable sized sliding windows decouple the arrivals from deletions—at any point in time *either* a new observation arrives or the oldest observation is deleted. A long string of arrivals increases the size of the window; a long string of deletions can reduce the size of the window. (We may sometimes exploit an upper bound on the size of the variable window, if such a bound is known in advance.)

Fixed Size Windows

We can implement a trivial fixed sliding window summary with precision ϵ by dividing the input stream into blocks of $\epsilon W/2$ consecutive observations. We summarize each block with an $\epsilon/2$ precision quantile summary. The block summarizing the most recent data is *under construction*. We add each new arrival to it until it contains $\epsilon W/2$ observations, and is considered *complete*. Once the block is complete, it is no longer modified.³ We store only the summaries of the last $2/\epsilon$ blocks. When a single observation in a block exits the window (it is “deleted”), we mark that block *expired*, and do not include it in our combined summary. These block summaries cover at most the last W , and at least the last $W - \epsilon W/2$, observations. By Corollary 1, the combined summary has a precision of $\epsilon/2$. If the most recent block has only just been started, and covers a very few observations, then our combined summary is missing up to $\epsilon W/2$ observations. In the worst case, all the missing observations have values that lie between our current estimate and the true ϕ quantile of interest. Even so, they could increase the error in rank by at most $\epsilon W/2$, keeping the total error below ϵW , ensuring that our combined sliding window summary has precision ϵ . The summary for each individual block uses $O(\frac{1}{\epsilon} \log(\epsilon^2 W))$ space, and the aggregate uses $O(\frac{1}{\epsilon^2} \log(\epsilon^2 W))$ space.

Arasu and Manku [2] employ the same basic structure of maintaining a set of summaries over fixed size windows in the input stream, but use a more sophisticated approach improves upon the space bound achieved by the simple algorithm above. Their algorithm suffers a blowup of only a factor of $O(\log(1/\epsilon))$ for maintaining a summary over a window of size W , compared to a blowup of $\Omega(1/\epsilon)$ in the simple implementation. When ϵ is small (say 0.001), this improvement is significant.

Arasu and Manku use a data structure with $L + 1$ levels where each level covers the stream by blocks of geometrically increasing sizes. At each point in time, exactly

³Lin et al. [13] use a variant of this simple approach, but construct $\epsilon W/4$ size blocks, and call prune on completed blocks.

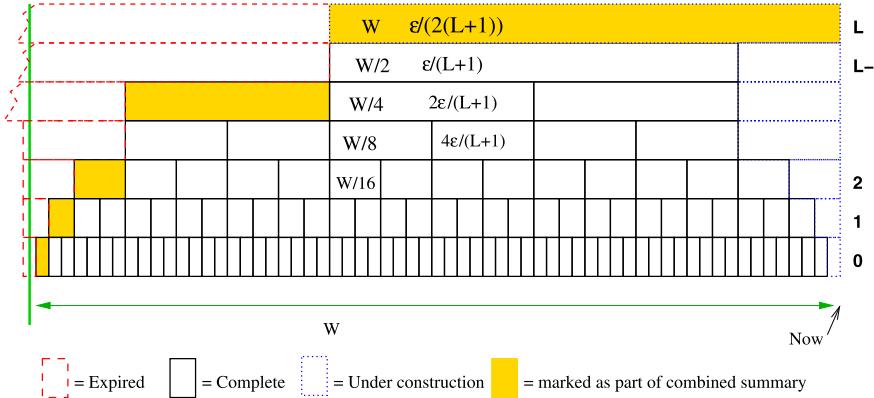


Fig. 8 Graphical representation of levels in Arasu–Manku algorithm

one block in each of the $L + 1$ levels is under construction. Each block is constructed using the GK algorithm until it is complete. It may seem that summarizing the entire window in $L + 1$ different ways, with $L + 1$ sets of blocks, would increase the required space, but, in fact, the total space requirement is *reduced* due to two basic observations.

First, once a block is complete, we can call `prune` to reduce the required space to $O(1/\epsilon)$. Second, for a given precision, larger blocks summarize the data stream more efficiently than smaller blocks—each stored tuple covers more observations. On the other hand, small blocks result in fewer lost observations when we discard the oldest block. The Arasu–Manku algorithm summarizes the window by combining non-overlapping blocks of *different* sizes. It summarizes most of the window, efficiently, using large blocks with fine precision (saving space due to the large block size). We can summarize the tail end of the window using coarser precision smaller blocks (saving space by the coarse precision), and bounding the number of lost observations at the very end of the window to the size of the smallest block we use.

Specifically, [2] divides the input stream into blocks in $L + 1$ different ways, where $L = \log_2(4/\epsilon)$. Each decomposition is called a *level*, and the levels are labeled 0 through L . As we go up each level the block size, denoted by N_ℓ , doubles to $2^\ell \epsilon W/4$, and the precision, denoted by ϵ_ℓ , is halved to $\frac{2^{(L-\ell)}\epsilon}{2(L+1)}$. This structure is graphically depicted in Fig. 8.

Proposition 7 *The Arasu–Manku window algorithm can summarize a fixed window of the W most recent observations, using at most $L + 1$ non-overlapping blocks from its summary structure.*

Proof The most recent observations are covered by the level L block currently under construction. For any $0 \leq \ell \leq L$, let η_ℓ denote the number of observations that need to be “covered” by blocks from level 0 through ℓ . We start by defining η_{L-1} to be

the number of observations not covered by a level L block; clearly, $\eta_{L-1} < W$. If $\eta_{L-1} \geq W/2$, then we can include a level $L - 1$ block of size $W/2$ to cover $W/2$ additional observations, and set $\eta_{L-2} = \eta_{L-1} - W/2 < W/2$. Otherwise, $\eta_{L-2} = \eta_{L-1} < W/2$. It is easy to see $\eta_{L-\ell} < W^{2^{-\ell+1}}$, and consequently at most one block from each level will be used. There are $L + 1$ levels. \square

Lemma 12 *The Arasu–Manku window algorithm implements an ϵ -approximate quantile-summary of a fixed window of the W most recent observations.*

Proof Let \mathbf{A} denote the combined summary of non-overlapping blocks. From the analysis in the proof of Lemma 1, the maximum gap between consecutive stored observations in \mathbf{A} is $\sum_{\ell=0}^L \hat{i}_\ell N_\ell \epsilon_\ell$, where \hat{i}_ℓ is 1 if a block from level ℓ is present in \mathbf{A} , and 0 if it is not. But

$$N_\ell \epsilon_\ell = 2^\ell \epsilon \left(\frac{W}{4} \right) 2^{L-\ell} \frac{\epsilon}{2(L+1)} = \epsilon \left(\frac{W}{4} \right) 2^L \frac{\epsilon}{2(L+1)},$$

a value that is independent of ℓ . Since $2^L = 4/\epsilon$, this can be simplified to $\frac{\epsilon W}{2(L+1)}$. Consequently, the maximum gap is simply $\frac{\epsilon W}{2(L+1)}$ times the number of blocks used in \mathbf{A} . By Proposition 7, there are at most $L + 1$ blocks in \mathbf{A} .

Therefore, the maximum gap is at most $(L + 1) \frac{\epsilon W}{2(L+1)}$, or $\epsilon W/2$.

An upper bound on the number of unsummarized observations in the window is just the smallest block size in the summary, namely $N_0 = \epsilon W/4$. Combining this with the precision of \mathbf{A} allows us to answer order-statistic queries with precision $3\epsilon/4$ (which is less than ϵ). \square

Lemma 13 *The Arasu–Manku algorithm can answer quantile queries with ϵ precision over a fixed window of W elements in $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log W)$ space.*

Proof There are at most $2^{L-\ell}$ complete blocks at each level, and those are pruned to $O(1/\epsilon_\ell)$ space. There are $L = O(\log 1/\epsilon)$ levels; the complete blocks therefore use

$$\sum_{\ell=1}^L \left(\frac{2^{L-\ell}}{\epsilon_\ell} \right) = \sum_{\ell=1}^L \left(\frac{2^{L-\ell}(2(2L+2))}{\epsilon 2^{L-\ell}} \right) = \frac{2L(2L+2)}{\epsilon} = O\left(\frac{L^2}{\epsilon}\right).$$

So the aggregate space used by the completed blocks at all levels is $O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon})$.

At each level ℓ , there is also at most one block that is still under construction. At its largest, just before it is completed, that block uses $O(\frac{1}{\epsilon_\ell} \log \epsilon_\ell N_\ell)$ space in the worst case. Expanding ϵ_ℓ (also, as noted above, $\epsilon_\ell N_\ell = \frac{\epsilon W}{2(L+1)}$) yields that the space used by a level ℓ block in construction is $O(2^{\ell-L} \frac{2(L+1)}{\epsilon} \log \frac{\epsilon W}{2(L+1)})$. Summing over all levels ℓ gives us a geometric series whose sum is

$$\begin{aligned}
O\left(\frac{2(L+1)}{\epsilon} \log \frac{\epsilon W}{2(L+1)}\right) &= O\left(\frac{2(\log \frac{4}{\epsilon} + 1)}{\epsilon} \log \frac{\epsilon W}{2(L+1)}\right) \\
&= O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log W\right).
\end{aligned}$$

The combined space is just the sum of the completed blocks and the blocks under construction, namely $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log W + \frac{1}{\epsilon} \log^2 \frac{1}{\epsilon})$, which can be simplified to $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log W)$ since the case when $W < \frac{1}{\epsilon}$ can be trivially solved using $O(\frac{1}{\epsilon})$ space. \square

Variable Size Windows

Although the algorithms described above assume that W is fixed, they can be extended in a straightforward way to handle variable sized windows. At any point in time we assume that the maximum window size is some W . If the actual window size differs from W by more than a factor of 2, then we alter our assumed window size to $2W$ or $W/2$, and update our data structures accordingly.

We first consider the case of a sliding window whose size has grown. We increase our assumed window size to $2W$. That this is generally possible follows from the calculation of the maximum gap between the minimum and maximum rank of two consecutive stored tuples in the combined summary. If we have an ϵ -approximate summary of a window of size W , then the maximum such gap is of size $2\epsilon W$. If the window size is increased to $2W$, then that data structure can answer queries to a precision of $\epsilon/2$.

In particular, for the trivial fixed window algorithm described earlier, as the window size grows, we do not alter the block size or the precision per block. We continue to add enough blocks of size $\epsilon W/2$ to summarize our entire window. When the number of blocks increases to $4/\epsilon$ (corresponding to a window size of $2W$), we simply merge every adjacent pair of blocks into a single block of size ϵW .

Extending the Arasu–Manku summary to accommodate a window that has grown beyond W is just as easy. Recall that the number of levels, L , is a function of ϵ and not W . So as W changes, the number of levels remain constant. However, a block at level l is now twice the size as before. We already have blocks of the required size of the new level l in the old level $l+1$ summary. Fortunately, those blocks are twice as precise as needed. To proceed with a new W of twice the size, then, we simply discard the level 0 blocks, and rename each level l as level $l-1$. To construct level L , the highest level, we simply duplicate the old level L . Although the nominal size of those blocks were of size W , the *expired* block is of no consequence, and the block that is *under construction* is truncated to the current window size—which is guaranteed to be less than or equal to $W-1$.

We now consider the case of a sliding window whose size has shrunk significantly. It is easy to see that we can accommodate any size window smaller than W by accepting a space blowup of at most a factor of $O(\log W)$. We can treat any

fixed window summary as a black box, and simultaneously maintain summaries for window sizes W , $W/2$, $W/4$, and so on. Each time the actual window size halves, we can discard the summary over the largest window.

Lin et al. [13] consider another variant of the sliding window model. In their “ n of N ” model, we once again summarize a fixed window consisting of the W latest observations. However, a query for the ϕ -quantile can be qualified by any integer $w \leq W$ such that the returned value must be the observation with rank $w\phi$ of the most recent w observations. It should be clear that the variable window extension to the Arasu–Manku algorithm must also be able to answer such queries with precision ϵ . If not, the algorithm would not be able to accurately answer subsequent quantile queries after the window shrank by $W - w$ consecutive deletions.

5.3 Distributed Quantile Summaries

The summary algorithms for streaming data described so far consider a setting in which the data can be observed at a centralized location. In such a setting, the GK algorithm is more space-efficient than algorithms in the `combine` and `prune` family.

However, in some settings the input stream is not observable at a single location. The aggregate quantile summary over a data set must then be computed by merging summaries over each of the subsets comprising that total set. For example, it may be desirable to split an input stream across different nodes in a large cluster, in order to process the input stream in parallel. Alternatively, in sensor networks whose nodes are organized into a tree, nodes may summarize the data in their subtree in order to avoid the communication costs of sending individual sensor readings to the root. In such cases, each site produces a summary of a subset of the stream and then passes the summary to another location where it is merged with summaries of other subsets until the entire stream is summarized. Algorithms that are built out of `combine` and `prune` can be extended naturally to such settings.

Nodes in sensor networks are typically resource-scarce. In particular, they have very limited memory and must conserve power. Summary algorithms over sensor networks must therefore optimize transmission cost between nodes (communication costs are the dominant drain on power). Nodes in sensor networks typically send their sensor readings up to the root through a tree-shaped topology. In order to reduce the transmission cost, each node may aggregate the data from their children and summarize it before passing it on. In this book, Chap. VI.1 (“Sensor Networks”, by Madden) describes sensor networks in more detail. Section 1.3 of [11] discusses the relation between streaming data and sensor networks. We briefly discuss here two summary algorithms over sensor networks (more details are available in the corresponding papers). Both Greenwald and Khanna [11] and Shrivastava et al. [18] support efficient ϵ -approximate quantile queries over sensor networks. Both can be characterized as applications of `combine`. One (see [11]) uses `prune` to manage communication costs, while the other (see [18]) uses a variant of COMPRESS.

A General Algorithm Using `combine` and `prune`

A simple algorithm to build quantile summaries over sensor networks collects the summaries of all the children of a node, and applies `combine` to produce a single summary that is sent to the parent. The operation `prune` is then applied to reduce the size of the data transmitted up the tree. Unfortunately, each application of `prune` to reduce the input to a buffer of size K can result in a loss of precision of up to $1/(2K)$. If the network topology is a tree of large depth, then either the aggregate loss of precision is too high, or else K must be very large and little reduction of communication costs can be achieved. Consequently, a slightly more complex strategy is required.

The general approach of the algorithms in [11] is to decouple the combining tree from the sensor network routing tree. The parent of a node in the combining tree may not be the immediate parent in the routing tree—rather the parent may be any ancestor at an arbitrary height up the tree. Physical nodes in the sensor network pass all summaries up through their parents in the routing tree until they hit the node that is considered the parent in the combining tree. The combining tree, and not the routing topology, controls the number of `combine` and `prune` operations executed by the algorithm. Intuitively, the goal of this algorithm is to build the widest, shallowest, such tree subject to minimizing the worst case amount of data sent from a child to its parent in the underlying physical topology. The paper shows that, regardless of topology, an ϵ -approximate quantile summary can be constructed over a sensor network with n nodes using a maximum per-node transmission cost of $O(\log^2 n/\epsilon)$. Let h , the “height” of the sensor network, denote the number of hops from the root of the network to the farthest leaf node. If h is known, then an embedding with a maximum per-node transmission cost of $O((\log n \log \frac{h}{\epsilon})/\epsilon)$ is achievable. Finally, if h is known to be smaller than $\log n$, then the maximum per-node transmission cost can be bounded by $O(h/\epsilon)$.

Optimized Algorithm When the Range of Values Is Known

Shrivastava et al. [18] study a slightly different setting in which observations can only take on integer values in the range $[1..M]$, where M is known in advance. Further, there is no requirement that the value v , returned by a quantile query, must be an element of S . Their algorithm is therefore able to pursue a different strategy. Rather than calling `prune`, which may lose precision on every call, it calls a variant of `COMPRESS`, which reduces the size of the summary as much as it can, while still preserving the precision. Thus their `COMPRESS`’ operation, which we will describe shortly, can be performed after each call to `combine`, at each node in the network.

The basic strategy is to construct a summary Q , called a *q-digest*, that consists of a sparse binary tree over the data range $1..M$. Each node b in the tree, referred to as a “bucket”, maintains a count $b.g$ that represents observations that fell between the minimum and maximum value of the bucket. Each bucket has two children, covering the lower and upper halves of its range. To keep the summary small, `COMPRESS`’

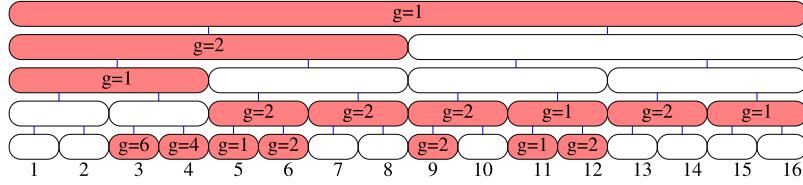


Fig. 9 An example q-digest, Q . Each non-empty bucket is labeled by g , the count of observations within that bucket. Q has $M = 16$, $n = 32$, and $\epsilon = 0.25$. It follows from these values that if the aggregate count in two siblings and their parent is $\leq (32 \times (0.25))/4 = 2$, then COMPRESS' will delete the two children and merge them into their parent. If n were 48, then the allowed aggregate count would be 3. In such a case, [1..4] could be merged into [1..8], and both [13..14] and [15..16] could be merged together into [13..16]. [9..10] and [11..12] would be merged into [9..12]. This would open up the subtree under [9..12]. After the next insertion (assuming it did not lie in the range [9..12]), then the 2 observations in [9] would be moved up into [9..10], and both [11] and [12] merged into [11..12]

moves observations in underpopulated buckets up the tree, to levels where larger ranges can be covered less precisely by fewer buckets. Buckets with a count of 0 are elided.

The combine operation when applied to a pair of such summaries simply sums the counts in corresponding buckets. Insertion of a new sensor value v into a summary Q is implemented by $\text{combine}(Q, Q')$, where Q' is a new summary consisting of only the single bucket $[v]$ with count = 1. COMPRESS' is called after each combine operation.

The precise behavior of COMPRESS' is a depth-first tree-walk starting at the leaves. It checks to see whether a bucket can be merged into its parent. When visiting a bucket (other than the root, which has no sibling or parent), it sums up the count of observations in the bucket, its sibling bucket, and its parent's bucket. If the bucket has a sum $\leq \lfloor \frac{\epsilon n}{\log M} \rfloor$, then COMPRESS' deletes the bucket, and adds the count to its parent.

Proposition 8 *The maximum count in any non-leaf node in Q is $\frac{\epsilon n}{\log M}$.*

Proof Non-leaf nodes only cover new observations by either COMPRESS' deleting their children, or by `combine` taking two q-digests and merging two corresponding buckets. The first case trivially maintains the bound because COMPRESS' will never delete a pair of children if the sum of the children and their parent exceeds $\frac{\epsilon n}{\log M}$. In the second case, when `combine` combines two q-digests containing n_1 and n_2 observations, respectively, the combined digest contains $n = n_1 + n_2$ observations. Prior to combine the two corresponding buckets contained fewer than $\epsilon n_1 / \log(M)$ and $\epsilon n_2 / \log(M)$ observations, respectively. After summing they contain fewer than $\frac{\epsilon(n_1+n_2)}{\log M} = \frac{\epsilon(n)}{\log M}$ observations, and the proposition holds. \square

We can use Q to answer quantile queries over S . The minimum rank of a value v in the data set S is computed by adding 1 to the cumulative counts in all of the buckets that contain only values less than v . The maximum rank of v is computed

by adding to the minimum rank, the sum of the counts in all non-leaf buckets whose minimum value is less than v , but whose maximum value is greater than or equal to v . For example, in Fig. 9, the minimum rank of the value 5 is 12, because 5 could be the first value after the 6 instances of 3, the 4 instances of 4, and the single observation that lies somewhere within [1,4]. The two observations within [1,8] and the single observation within [1,16] *may* be values that are greater than or equal to 5, and hence cannot be counted in the minimum possible rank. On the other hand, there *may* be values that precede 5, and hence could increase the rank of 5 to be as high as 15.

Theorem 6 *A q -digest $Q(\epsilon)$ summarizing a dataset S is an ϵ -approximate quantile summary using at most $3 \log(M)/\epsilon$ buckets, regardless of the size of $|S|$.*

We first establish the upper bound on the size of Q . Let k denote the number of surviving non-zero buckets in Q . For $b \in Q$, let $b.g$ be the count of observations in b , and let $s(b)$ denote the sibling of bucket b in Q , and $p(b)$ denote its parent in Q . Every non-empty bucket b obeys $b.g + s(b).g + p(b).g > \frac{\epsilon n}{\log M}$. Each bucket can appear at most once as a left sibling, once as a right sibling, and at most once as a parent. Summing this inequality over all k surviving buckets, we get $3n \geq \sum_{b \in Q} (b.g + s(b).g + p(b).g) > k \frac{\epsilon n}{\log M}$. Therefore, $3 \log(M)/\epsilon > k$.

We next show that $Q(\epsilon)$ is an ϵ -approximate quantile summary. We observe that at most one bucket at each level of the tree can overlap the leaf bucket containing v . The height of the tree (excluding the leaves) is $\log M$, and by Proposition 8, each non-leaf bucket contains at most $\frac{\epsilon n}{\log M}$ observations, so the cumulative gap between minimum and maximum rank is at most ϵn . The biggest gap between the minimum rank of a stored value v in Q and the maximum rank of its successor, v' , is therefore $2\epsilon n$, and by Proposition 1, Q is an ϵ -approximate quantile summary.

6 Concluding Remarks

We presented here a broad range of algorithmic ideas for computing quantile summaries of data streams using small space. We highlighted connections among these ideas, and how techniques developed for one setting sometimes naturally lend themselves to a seemingly different setting. While the past decade has seen significant advances in space-efficient computation of quantile summaries, some fundamental questions remain unresolved. For instance, in the cash-register model, it is not known if the space bound of $O(\log(\epsilon n)/\epsilon)$ achieved by the GK algorithm [10] on a stream of length n is the best possible for any deterministic algorithm. When the elements are known to be in the range of $[1..M]$ for some positive integer M , is the $O(\log(M)/\epsilon)$ bound achieved by the q -digest algorithm [18] optimal? In either setting, only a trivial lower bound of $\Omega(1/\epsilon)$ on space is known. Similarly, when randomization is allowed, what is the best possible dependence of space needed on ϵ and δ ? It appears that progress on these questions would require significant new ideas that may help advance our understanding of space-bounded computation as a whole.

References

1. K. Alsabti, S. Ranka, V. Singh, A one-pass algorithm for accurately estimating quantiles for disk-resident data, in *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, ed. by M. Jarke et al., Los Altos, CA 94022, USA (Morgan Kaufmann, San Mateo, 1997), pp. 346–355
2. A. Arasu, G.S. Manku, Approximate counts and quantiles over sliding windows, in *Proceedings of the 23rd ACM Symposium on Principles of Database Systems (PODS 2004)*, Paris, France (2004), pp. 286–296
3. J.L. Carter, M.N. Wegman, Universal classes of hash functions, in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, ed. by M. Jarke et al., Los Altos, CA 94022, USA (ACM, Colorado, 1977), pp. 106–112
4. G. Cormode, S. Muthukrishnan, An improved data stream summary: the Count-Min sketch and its applications, in *Proceedings of Latin American Theoretical Informatics (LATIN'04)* (2004)
5. P.B. Gibbons, Y. Matias, V. Poosala, Fast incremental maintenance of approximate histograms, in *Proc. 23rd Int. Conf. Very Large Data Bases, VLDB*, ed. by M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, M.A. Jeusfeld (Morgan Kaufmann, San Mateo, 1997), pp. 466–475
6. P.B. Gibbons, Y. Matias, V. Poosala, Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.* **27**(3), 261–298 (2003)
7. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *Proceedings of the 27th Intl. Conf. Very Large Data Bases, VLDB*, ed. by P.M.G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, R.T. Snodgrass, Rome, Italy (2001), pp. 79–88
8. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, How to summarize the universe: dynamic maintenance of quantiles, in *Proceedings of the 28th Intl. Conf. Very Large Data Bases, VLDB*, Hong Kong, China (2002), pp. 454–465
9. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, Domain-driven data synopses for dynamic quantiles. *IEEE Trans. Knowl. Data Eng.* **17**(7), 927–938 (2005)
10. M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in *Proceedings of the 2001 ACM SIGMOD Intl. Conference on Management of Data* (2001), pp. 58–66
11. M.B. Greenwald, S. Khanna, Power-conserving computation of order-statistics over sensor networks, in *Proceedings of the 23rd ACM Symposium on Principles of Database Systems (PODS 2004)*, Paris, France (2004), pp. 275–285
12. S. Guha, A. McGregor, Lower bounds for quantile estimation in random-order and multi-pass streaming, in *International Colloquium on Automata, Languages and Programming* (2007)
13. X. Lin, H. Lu, J. Xu, J.X. Yu, Continuously maintaining quantile summaries of the most recent n elements over a data stream, in *Proceedings of the 20th International Conference on Data Engineering (ICDE04)*, Boston, MA (IEEE Comput. Soc., Los Alamitos, 2004), pp. 362–374
14. G. Singh Manku, S. Rajagopalan, B.G. Lindsay, Approximate medians and other quantiles in one pass and with limited memory. *SIGMOD Rec.* **27**(2), 426–435 (1998) (ACM Special Interest Group on Management of Data) SIGMOD'98, Seattle, WA
15. G. Singh Manku, S. Rajagopalan, B.G. Lindsay, Random sampling techniques for space efficient online computation of order statistics of large datasets, *SIGMOD Rec.* **28**(2), 251–262 (1999) (ACM Special Interest Group on Management of Data) SIGMOD'99, Philadelphia, PA
16. J.I. Munro, M.S. Paterson, Selection and sorting with limited storage. *Theor. Comput. Sci.* **12**, 315–323 (1980)

17. S. Muthukrishnan, Data streams: algorithms and applications (2003). Unpublished report (invited talk at SODA03), available at <http://athos.ruthers.edu/~muthu/stream-1-1.ps>
18. N. Shrivastava, C. Buragohain, D. Agrawal, S. Suri, Medians and beyond: new aggregation techniques for sensor networks, in *Proceedings of the 2nd ACM Conference on Embedded Network Sensor Systems (SenSys'04)*, Baltimore, MD (2004), pp. 239–249

Join Sizes, Frequency Moments, and Applications

Graham Cormode and Minos Garofalakis

1 Introduction

In this chapter, we focus on a set of problems chiefly inspired by the problem of estimating the *size of the (equi-)join* between two relational data streams. This problem is at the heart of a wide variety of other problems, both in databases/data streams and beyond. Given two relations, R and S , and an attribute a common to both relations, the *equi-join* between the two relations, $R \bowtie S$, consists of one tuple for each $r \in R$ and $s \in S$ pair such that $r.a = s.a$. Estimating the join size between pairs of relations is a key component in designing an efficient execution plan for a complex SQL query that may contain arbitrary combinations of selection, projection, and join tasks; as such, it forms a critical part of database *query optimization* [15]. The results can also be employed to provide *fast approximate answers* to user queries, to allow, for instance, interactive exploration of massive data sets [12]. The ideas discussed in this chapter are directly applicable in a streaming context, and, additionally, within traditional database management systems where: (i) relational tables are *dynamic*, and queries must be tracked over the stream of updates generated by tuple insertions and deletions; or, (ii) relations are truly massive, and *single-pass algorithms* are the only viable option for efficient query processing. Knowing the join size is also an important problem at the heart of a variety of other problems, such as building histogram and wavelet representations of data, and can be applied

G. Cormode (✉)

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
e-mail: G.Cormode@warwick.ac.uk

M. Garofalakis

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

to problems such as finding frequent items and quantiles, and modeling spatial and high-dimensional data.

Many problems over streams of data can be modeled as problems over (implicit) vectors which are defined incrementally a continuous stream of data updates. For example, given a stream of communications between pairs of people (such as records of telephone calls between a caller and callee), we can capture information about the stream in a vector, indexed by the (number of the) calling party, and recording, say, the number of calls made by that caller. Each new call causes us to update one entry in this vector (i.e., incrementing the corresponding counter). Queries about specific calling patterns can often be rendered into queries over this vector representation: For instance, to find the number of distinct callers active on the network, we must count the number of non-zero entries in the vector. Similarly, problems such as computing the number of calls made by a particular number (or, range of numbers), the number of callers who have made more than 50 calls, the median number of calls made and so on, can all be posed as function computations over this vector. In an even more dynamic setting, such streaming vectors could be used to track the number of *active* TCP connections in a large Internet Service Provider (ISP) network (say, per source IP address); thus, a TCP-connection open (close) message would have to increment (respectively, decrement) the appropriate counter in the vector.

Translating the join-size problem into the vector setting, we observe that each relation can essentially be represented by a *frequency-distribution vector*, whose i th component counts the number of occurrences of join-attribute value i in the relation. (Without loss of generality, we assume the join attributes to range over an integer domain $[U] = \{1, \dots, U\}$, for some large U .¹) The join size $|R \bowtie S|$ between two streaming relations R and S corresponds to computing the *inner-product* of their frequency-distribution vectors, that is, the sum of the product of the counts of i in R and S over all $i \in [U]$. A special case of this query is the *self-join size* $|R \bowtie R|$: the size of the join between a relation R and itself. This is the sum of the squares of the entries in the corresponding frequency-distribution vector, or, equivalently, the square of its Euclidean (i.e., L_2) norm.

The challenge for designing effective and scalable streaming solutions for such problems is that it is not practical to materialize this vector representation; both the size of the input data stream and the size of the “universe” from which items in the stream are drawn can grow to be very large indeed. Thus, naive solutions that employ $O(U)$ space or time over the update stream are not feasible. Instead, we adopt a paradigm of based on *approximate, randomized estimation algorithms* that can produce answers to such queries with provable guarantees on the accuracy of the approximation while using only small space and time per streaming update.

Beyond join-size approximations, efficiently estimating such vector inner-products and norms has a wide variety of applications in streaming computation problems, including approximating range-query aggregates, quantiles, and heavy-hitter elements, and building approximate histograms and wavelet representations.

¹While our development here assumes a known upper bound on the attribute universe size U , this is not required: U can be learned adaptively over the stream using standard tricks (see, e.g., [13]).

We briefly touch upon some of these applications later in this chapter, while later chapters also provide more detailed treatments of specific applications of the techniques. Our discussion in this chapter focuses on efficient, *sketch-based* streaming algorithms for join-size and self-join-size estimation problems, based on two influential papers by Alon, Matias, and Szegedy [3], and Alon, Gibbons, Matias, and Szegedy [2]. The remainder of the chapter is structured as follows.

2 Preliminaries and Problem Formulation

Let \mathbf{x} be a (large) vector being defined incrementally by a stream of data updates. We adopt the most general (so-called, “*turnstile*” [20]) model of streaming data, where each update in the stream is a pair (i, c) , where $i \in [U]$ is the index of the entry being updated and c is a positive or negative number denoting the magnitude of the update; in other words, the (i, c) update sets $\mathbf{x}[i] = \mathbf{x}[i] + c$. Allowing c values to be either positive or negative implies that \mathbf{x} vector entries can decrease as well as increase. Thus, this model allows us to easily represent the “departure” of items (e.g., closed TCP connections) as well as their arrival.

Definition 1 (Join Size) The *(equi-)join size* of two streaming relations with frequency-distribution vectors \mathbf{x} and \mathbf{y} (over universe $[U]$) is exactly the *inner product* of \mathbf{x} and \mathbf{y} , defined as $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^U \mathbf{x}[i]\mathbf{y}[i]$.

(We use the terms “join size” and “inner product” interchangeably in the remainder of this chapter.) The special case of *self-join size* (i.e., inner-product of a vector with itself) is closely related to the notion of *frequency moments* of a data distribution, which can be defined using the same streaming model and concepts. More specifically, let \mathbf{x} denote the frequency-distribution vector for a stream R of items (from domain $[U]$); that is, $\mathbf{x}[i]$ denotes the number of occurrences of item i in the R stream. Then, the *pth frequency moment* of stream R is given by $F_p(\mathbf{x}) = \sum_{i=1}^U \mathbf{x}[i]^p$.

Observe that F_1 is simply the length of the stream R (i.e., the total number of observed items), and can be easily computed with a single counter. F_0 is the number of distinct items in the R stream (the size of the *set* of items that appear), and is the focus of other chapters in this volume. F_2 , the second frequency moment, is also known as the *repeat rate* of the sequence, or as “*Gini’s index of homogeneity*”—it forms the basis of a variety of data analysis tasks, and can be used to compute the *surprise index* [14] and the *self-correlation* of the stream.

We extend the definition of frequency moments to the arrival vector model. Now, $F_p(\mathbf{x}) = \sum_{i=1}^U \mathbf{x}[i]^p$; thus, $F_p(\mathbf{x}) = \|\mathbf{x}\|_p^p$, where $\|\cdot\|_p$ denotes the L_p vector norm (see also the chapter of Cormode and Indyk later in this volume). In particular, $F_2(\mathbf{x})$ is the self-join size of a relation whose characteristic vector is \mathbf{x} , and $\sqrt{F_2(\mathbf{x})}$ is the L_2 , or Euclidean, norm of the vector \mathbf{x} .

We give the definition of other relevant functions:

Definition 2 The *vector distance* between two vectors \mathbf{x} and \mathbf{y} , both of dimensionality U is given by $\|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{F_2(\mathbf{x} - \mathbf{y})}$.

Definition 3 An (ϵ, δ) -relative approximation of a value X returns an answer x such that, with probability at least $1 - \delta$,

$$(1 - \epsilon)X \leq x \leq (1 + \epsilon)X.$$

Definition 4 (k-Wise Independent Hash Functions) A family of hash functions \mathcal{H} mapping items from X onto a set Y is said to be *k-wise independent* if, over random choices of $h \in \mathcal{H}$, we have

$$\Pr[h(x_1) = h(x_2) = \dots = h(x_k)] = \frac{1}{|Y|^k}.$$

In other words, for up to k items, we can treat the results of the hash function as independent random events, and reason about them correspondingly. Here, we will make use of families of 2-wise (pairwise) independent hash functions, and 4-wise independent hash functions. Such hash functions are easy to implement: the family $\mathcal{H}_2 = \{ax + b \bmod P \bmod |Y|\}$, where P is a prime and a and b are picked uniformly at random from $\{0, \dots, P - 1\}$ is pairwise independent onto $\{0, \dots, |Y| - 1\}$ [5, 19]. More generally, the family

$$\mathcal{H}_k = \left\{ \sum_{i=0}^{k-1} c_i x^i \bmod P \bmod |Y| \right\}$$

is k -wise independent for c_i 's picked uniformly from $\{0, \dots, |Y| - 1\}$ [23]. Various efficient implementations of such functions have been given, especially for the case of $k = 2$ and $k = 4$ [21, 22].

3 AMS Sketches

In their 1996 paper, Alon, Matias and Szegedy gave an algorithm to give an (ϵ, δ) -approximation of the self-join size. The algorithm computes a data structure, where each entry in the data structure is computed through an identical procedure but with a different 4-wise independent hash function for each entry. Each entry can be used to find an estimate of the self-join size that is correct in expectation, but can be far from the correct value. Carefully combining all estimates gives a result that is an (ϵ, δ) -approximation as required. The resulting data structure is often called an *AMS* (or, “tug-of-war”) *sketch*, since the data structure concisely summarizes, or ‘sketches’ a much larger amount of information.

<pre>UPDATE(i, c, z) Input: item i, count c, sketch z 1: for $j = 1$ to w do 2: for $k = 1$ to d do 3: $z[j][k] += h_{j,k}(i) * c$</pre>	<pre>ESTIMATEJS(x, y) Input: sketch x, sketch y Output: estimate of $\mathbf{x} \cdot \mathbf{y}$ 1: for $j = 1$ to w do 2: $avg[j] = 0$; 3: for $k = 1$ to d do 4: $avg[j] += x[j][k] * y[j][k] / w$; 5: Return(median(avg))</pre>
<pre>ESTIMATE$F_2(z)$ Input: sketch z 1: Return ESTIMATEJS(z, z)</pre>	

Fig. 1 AMS algorithm for estimating join and self-join size

To build one element of the sketch, the algorithm takes a 4-wise hash function $h : [1..U] \rightarrow \{-1, +1\}$ and computes $Z = \sum_{i=1}^U h(i)\mathbf{x}[i]$. Note that this is easy to maintain under the turnstile streaming model: initialize $Z = 0$, and for every update in the stream (i, c) set $Z = Z + c * h(i)$. This algorithm is given in pseudocode as UPDATE in Fig. 1.

3.1 Second Frequency Moment Estimation

To estimate the self-join size, we compute Z^2 .

Lemma 1 $\mathbb{E}(Z^2) = F_2(\mathbf{x})$

Proof

$$\begin{aligned} \mathbb{E}(Z^2) &= \mathbb{E}\left(\left(\sum_{i=1}^U h(i)\mathbf{x}[i]\right)^2\right) \\ &= \mathbb{E}\left(\sum_{i=1}^U h(i)^2\mathbf{x}[i]^2\right) + \mathbb{E}\sum_{1 \leq i < j \leq U} 2h(i)h(j)\mathbf{x}[i]\mathbf{x}[j] \\ &= \sum_{i=1}^U \mathbf{x}[i]^2 + 0 = F_2(\mathbf{x}). \end{aligned}$$

The proof relies critically on the properties of h : $h(i)^2 = 1$ for all i , but since h is 4-wise independent then the outcomes $h(i) = h(j)$ and $h(i) = -h(j)$ are equally likely (for $j \neq i$) and so in expectation $h(i)h(j)$ is zero. \square

Lemma 2 $\text{Var}(Z^2) \leq 2F_2(\mathbf{x})^2$.

Proof

$$\begin{aligned}
\text{Var}(Z^2) &= \mathbb{E}(Z^4) - \mathbb{E}(Z^2)^2 \\
&= \mathbb{E}\left(\left(\sum_{i=1}^U h(i)\mathbf{x}[i]\right)^4\right) - \left(\sum_{i=1}^U \mathbf{x}[i]^2\right)^2 \\
&= \mathbb{E}\left(\left(\sum_{i=1}^U h(i)^4\mathbf{x}[i]^4 + \sum_{1 \leq i < j \leq U} 6h(i)^2h(j)^2\mathbf{x}[i]^2\mathbf{x}[j]^2\right.\right. \\
&\quad + \sum_{i,i \neq j \neq k} 12h(i)^2h(j)h(k)\mathbf{x}[i]^2\mathbf{x}[j]\mathbf{x}[k] \\
&\quad + \sum_{1 \leq i \neq j \leq U} 4h^3(i)h(j)\mathbf{x}[i]^3\mathbf{x}[j] \\
&\quad \left.\left.+ \sum_{1 \leq i < j < k < l \leq U} 12h(i)h(j)h(k)h(l)\mathbf{x}[i]\mathbf{x}[j]\mathbf{x}[k]\mathbf{x}[l]\right)\right. \\
&\quad \left.- \left(\sum_{i=1}^U \mathbf{x}[i]^4 + \sum_{1 \leq i < j \leq U} 2\mathbf{x}[i]^2\mathbf{x}[j]^2\right)\right) \\
&= \sum_{i=1}^U \mathbf{x}[i]^4 + \sum_{1 \leq i < j \leq U} 6\mathbf{x}[i]^2\mathbf{x}[j]^2 \\
&\quad - \left(\sum_{i=1}^U \mathbf{x}[i]^4 + \sum_{1 \leq i < j \leq U} 2\mathbf{x}[i]^2\mathbf{x}[j]^2\right) \\
&= 4 \sum_{1 \leq i < j \leq U} \mathbf{x}[i]^2\mathbf{x}[j]^2 \leq 2F_2^2. \quad \square
\end{aligned}$$

This shows that each estimate is correct in expectation and has bounded variance. Again, in expectation many of the cross-terms (e.g., $h(i)h(j)h(k)h(l)$) are zero, by the 4-wise independence of the hash function h . In order to give tight guarantees about the accuracy of this procedure, we make use of a few statistical results about the average and median of random variables.

Fact 1 (Variance Reduction) *Let X_i be independent and identically distributed random variables. Then*

$$\text{Var}\left(\sum_{i=1}^w \frac{X_i}{w}\right) = \frac{1}{w} \text{Var}(X_1).$$

In other words, taking the average of w copies of an estimator reduces the variance by a factor of w .

Fact 2 (The Chebyshev Inequality) *Given a random variable X ,*

$$\Pr[|X - \mathbb{E}(X)| \geq k] \leq \frac{\text{Var}(X)}{k^2}.$$

Fact 3 (Application of Chernoff Bounds) *Let R be a range of values $R = [R_{\min}..R_{\max}]$, and let Y_i be $d = 4 \log 1/\delta$ independent and identically distributed random variable such that $\Pr[Y_i \notin R] \leq \frac{1}{8}$. Then*

$$\Pr[(\text{median}_{i=1}^d Y_i) \notin R] \leq \delta,$$

that is, if there is constant probability that each Y_i falls within the desired range R , then taking the median of $O(\log 1/\delta)$ copies of Y_i reduces the failure probability to δ .

For details of these facts, see a standard text such as [19]. For the final fact, observe that we can define an indicator variable for each Y_i that is 0 if Y_i falls within the range R and is 1 otherwise. The expectation of the sum of these indicator variables is $\frac{1}{2} \log 1/\delta$. However, if the median of the Y_i 's is not within range, then at least half the Y_i 's must have fallen outside the range; hence the sum of the indicator variables must be at least $2 \log 1/\delta$. Applying Chernoff bounds gives the derived result.

We can now apply these facts to show the accuracy of the estimation procedure for F_2 :

Theorem 1 *An (ϵ, δ) -approximation of F_2 , the self-join size, can be computed in space $O(\frac{1}{\epsilon^2} \log 1/\delta)$ machine words in the streaming model. Each update takes time $O(\frac{1}{\epsilon^2} \log 1/\delta)$.*

Proof Applying the Chebyshev inequality to the average of $w = \frac{16}{\epsilon^2}$ copies of the estimate Z generates a new estimate Y such that

$$\Pr[|Y - F_2| \leq \epsilon F_2] \leq \frac{\text{Var}(Y)}{\epsilon^2 F_2^2} = \frac{\text{Var}(Z)}{c \epsilon^2 F_2^2} = \frac{2F_2^2}{(16/\epsilon^2)\epsilon^2 F_2^2} = \frac{1}{8}.$$

Hence, applying the Chernoff bound result from Fact 3 to the median of $4 \log 1/\delta$ copies of the average Y gives the probability of the results being outside the range of ϵF_2 from F_2 as δ . The space required is that to maintain $O(\frac{1}{\epsilon^2} \log 1/\delta)$ copies of the original estimate. Each of these requires a counter and a 4-wise independent hash function, both of which can be represented with a constant number of machine words under the standard RAM model. \square

In summary, this shows that an (ϵ, δ) -approximation of F_2 can be computed using space that is essentially independent of the size of the stream or the dimensionality of the vector. The complete algorithm is given in Fig. 1.

3.2 Vector Difference Estimation

The results for self-join size estimation can be applied to the problem of measuring the distance between two vectors. The result follows almost as an immediate corollary of the previous theorem, combined with the structure of the sketch. Observe that the difference between \mathbf{x} and \mathbf{y} as given in Definition 2 can be thought of as the self-join size of a single vector whose i th entry is $\mathbf{x}[i] - \mathbf{y}[i]$. The sketch of this vector is given by $\sum_i h(i)(\mathbf{x}[i] - \mathbf{y}[i])$. This can be rewritten as $\sum_i h(i)\mathbf{x}[i] - \sum_i h(i)\mathbf{y}[i]$. In other words, the sketch of the difference is the difference of the sketches. Thus, by subtracting the sketches and then applying the estimation procedure we can get an (ϵ, δ) -approximation of the difference between the vectors.

This relies on the *linearity* of the sketching operation: any linear transformation (scaling, addition, subtraction, etc.) to the original vector can be applied on the sketch and the result is the sketch of the modified vector. This was used implicitly to show that the sketch can be updated dynamically under streaming updates. Such linearity properties have also been used in a variety of techniques for streaming data based on sketches. See the discussion in Sect. 4 for some examples.

3.3 Join Size Estimation

Lemma 3 *Let Z_x be an entry of a sketch computed for the vector \mathbf{x} , and let Z_y be an entry of a sketch computer for \mathbf{y} using the same hash function. The estimate is correct in expectation, i.e., $\mathbb{E}(Z_x * Z_y) = \mathbf{x} \cdot \mathbf{y}$.*

Proof

$$\begin{aligned}\mathbb{E}(Z_x * Z_y) &= \mathbb{E}\left(\sum_{i=1}^U h(i)^2 \mathbf{x}[i] \mathbf{y}[i] + \sum_{1 \leq i \neq j \leq U} h(i)h(j) \mathbf{x}[i] \mathbf{y}[j]\right) \\ &= \sum_{i=1}^U \mathbf{x}[i] \mathbf{y}[i] + 0 = \mathbf{x} \cdot \mathbf{y}.\end{aligned}$$

□

Lemma 4 $\text{Var}(Z_x * Z_y) \leq F_2(\mathbf{x})F_2(\mathbf{y})$.

Proof

$$\begin{aligned}
\text{Var}(Z_x * Z_y) &= \mathbb{E}(Z_x^2 Z_y^2) - \mathbb{E}(Z_x Z_y)^2 \\
&= \mathbb{E}\left(\sum_{i=1}^U h(i)^4 \mathbf{x}[i]^2 \mathbf{y}[i]^2 + \sum_{1 \leq i \neq j \leq U} h(i)^2 h(j)^2 \mathbf{x}[i]^2 \mathbf{y}[j]^2\right. \\
&\quad \left.+ \sum_{1 \leq i < j \leq U} 4h(i)^2 h(j)^2 \mathbf{x}[i] \mathbf{y}[i] \mathbf{x}[j] \mathbf{y}[j]\right) - (\mathbf{x} \cdot \mathbf{y})^2 \\
&= \sum_{i=1}^U (\mathbf{x}[i] \mathbf{y}[i])^2 + \sum_{1 \leq i \neq j \leq U} (\mathbf{x}[i] \mathbf{y}[j])^2 \\
&\quad + \sum_{1 \leq i < j \leq U} 4\mathbf{x}[i] \mathbf{y}[i] \mathbf{x}[j] \mathbf{y}[j] \\
&\quad - \left(\sum_{1 \leq i \leq U} (\mathbf{x}[i] \mathbf{y}[i])^2 + \sum_{1 \leq i < j \leq U} (2\mathbf{x}[i] \mathbf{y}[i] \mathbf{x}[j] \mathbf{y}[j])\right) \\
&\leq \sum_{1 \leq i < j} (\mathbf{x}[i] \mathbf{y}[j])^2 + 2 \sum_{1 \leq i < j \leq U} (2\mathbf{x}[i] \mathbf{y}[i] \mathbf{x}[j] \mathbf{y}[j]) \\
&\leq \sum_{i=1}^U \mathbf{x}[i]^2 \sum_{j=1}^U \mathbf{y}[j]^2 + \left(\sum_{i=1}^U \mathbf{x}[i] \mathbf{y}[i]\right)^2 \\
&\leq 2 \sum_{i=1}^U \mathbf{x}[i]^2 \sum_{j=1}^U \mathbf{y}[j]^2 = 2F_2(\mathbf{x})F_2(\mathbf{y}). \quad \square
\end{aligned}$$

Applying the Chebyshev Inequality to the average of $w = \frac{16}{\epsilon^2}$ copies of this estimate, and then taking the median of $d = 4 \log 1/\delta$ such averages, as in the proof of Theorem 1, allows us to state the following theorem:

Theorem 2 *Using space $O(\frac{1}{\epsilon^2} \log 1/\delta)$ space we can output an estimate of $\mathbf{x} \cdot \mathbf{y}$ so that*

$$\Pr[|(\mathbf{x} \cdot \mathbf{y}) - \text{est}| \leq \epsilon \sqrt{F_2(\mathbf{x})F_2(\mathbf{y})}] \geq 1 - \delta.$$

Note that for the special case of when $\mathbf{x} = \mathbf{y}$, then the above Theorem 2 reduces to Theorem 1. However, this is not an (ϵ, δ) -approximation since in general $\sqrt{F_2(\mathbf{x})F_2(\mathbf{y})} > (\mathbf{x} \cdot \mathbf{y})$. In order to get such an approximation, we need to increase w by a factor of $(\mathbf{x} \cdot \mathbf{y})^2 / (F_2(\mathbf{x})F_2(\mathbf{y}))$. This may be possible if we have *a priori* bounds on these quantities, but since we are trying to approximate $\mathbf{x} \cdot \mathbf{y}$, we cannot know this quantity exactly in advance. In general, we cannot hope for much stronger results due to the following negative result:

Theorem 3 *Guaranteeing an (ϵ, δ) -approximation of $\langle \mathbf{x} \cdot \mathbf{y} \rangle$ requires $\Omega(U)$ space in the worst case.*

Proof We reduce from the problem of testing whether two sets have any element in common, and use the communication complexity of this problem to argue a space bound for the streaming problem.

Consider two arbitrary sets X and Y , both of which are subsets of $[1..U]$. There are two people who wish to collaborate to compute a function of X and Y : X is held by one party and Y by the other. A well-known result from communication complexity states that determining whether there exists i such that $i \in X \wedge i \in Y$ requires communication between the two parties that is linear in U , even under a probabilistic model [18]. This is known as the disjointness problem, since the answer is either that the sets are disjoint (i.e., $X \cap Y = \emptyset$) or not disjoint.

First, we show that if we can approximate join size, then we can answer disjointness queries. Let $\mathbf{x}[i] = 1 \iff i \in X$, and zero otherwise; similarly, let $\mathbf{y}[j] = 1 \iff j \in Y$, otherwise $\mathbf{y}[j] = 0$. Now observe that $(X \cap Y = \emptyset) \iff (\mathbf{x} \cdot \mathbf{y} = 0)$. Hence, computing the join size exactly means that we can answer disjointness queries. More strongly, any approximation of $\mathbf{x} \cdot \mathbf{y}$ also allows us to answer disjointness queries, since to approximate $\mathbf{x} \cdot \mathbf{y} = 0$ we must output ‘0’, and if $\mathbf{x} \cdot \mathbf{y} \neq 0$, then no approximation can correctly output ‘0’. Thus any algorithm that approximates $\mathbf{x} \cdot \mathbf{y}$ must use $\Omega(U)$ bits of communication if \mathbf{x} is held by one party and \mathbf{y} by the other.

Now, we show how this bound applies to the streaming context. Suppose all of \mathbf{x} arrives in the stream first, and then \mathbf{y} arrives in the stream next. Consider the state (memory contents) held by any streaming algorithm to approximate $\mathbf{x} \cdot \mathbf{y}$ after \mathbf{x} has been seen. Imagine sending all the state to another copy of the algorithm which then receives the stream \mathbf{y} . If the algorithm correctly approximates $\mathbf{x} \cdot \mathbf{y}$, then the size of the data communicated must be $\Omega(U)$ bits. Hence, the space used by the algorithm must be at least $\Omega(U)$ bits. \square

Nevertheless, the results obtained by this estimation procedure in practice often give very good estimates of the join size between relations. In particular, it tends to significantly outperform solutions based on sampling, which do not give the guarantee of being correct in expectation.

4 Applications and Extensions

The generality of the join size aggregate and the simplicity and flexibility of the sketching technique, means that the sketching method has been used as the basis of a wide variety of other streaming algorithms. Rather than attempt to give a comprehensive survey of such techniques, we outline a few examples to illustrate the ways that this data structure has been applied and modified.

4.1 Point Estimation, Range Queries and Wavelets

The problem of point estimation is, given a vector \mathbf{x} specified as a data stream, to accept queries which specify a particular entry, i , and to return an estimate of $\mathbf{x}[i]$. Clearly, one cannot give exact answers, or guarantee very fine accuracy since to do so would allow one to recover the whole vector in the worst case. However, they can be well approximated as a corollary of the previous theorem.

Corollary 1 *Point queries can be answered using the same sketch structure in space $O(\frac{1}{\epsilon^2} \log 1/\delta)$ with error less than $\epsilon \sqrt{F_2(\mathbf{x})}$ with probability at least $1 - \delta$.*

Proof Observe that a point query can be specified as a join size query, $\mathbf{x} \cdot I_i$, where $I_i[i] = 1$, and $I_i[j] = 0$ for $j \neq i$. Applying Theorem 2, we find that we can answer point-estimation queries with error at most $\epsilon \sqrt{F_2(\mathbf{x})}$ with probability at least $1 - \delta$. \square

This is quite a strong guarantee, since typically we make require accuracy in terms of $\epsilon F_1(\mathbf{x})$, and for any \mathbf{x} , $\sqrt{F_2(\mathbf{x})} \leq F_1(\mathbf{x})$. Point Estimation is at the heart of many algorithms for finding “heavy hitters”: items in the data stream which occur very frequently. (This is explored further in a chapter by Charikar later in this volume.)

In a similar way, one can answer arbitrary *range queries* of the form $R(i, j) = \sum_{k=i}^j \mathbf{x}[k]$ by reducing this to an inner product with a vector I_i^j , where $I_i^j[k] = 1 \iff i \leq k \leq j$ and zero elsewhere. However, the error increases linearly with $|j - i + 1|$, making the accuracy too weak for large ranges. A standard approach is then to decompose any arbitrary range into at most $O(\log U)$ *dyadic ranges*, each of which has length a power of two, and begins at a multiple of its own length. Each dyadic range query can be treated as a point query, and hence arbitrary range queries can be answered to the same accuracy as point queries, with a blow-up in space polynomial in $\log U$ (see, for example, [13]).

Computing approximate (Haar) wavelet coefficients and approximate histogram summaries [9] can also make use of point, range and related queries, and hence techniques to approximate the wavelet coefficients of a signal presented in a streaming fashion make extensive use of sketch data structures. (More details of the methods used and the results obtained are given in a chapter by Muthukrishnan and Strauss later in this volume.)

4.2 Faster Implementations Using Hashing

One potential disadvantage of the sketching scheme is the time cost to process each update. For $w = O(\frac{1}{\epsilon^2})$ and $d = O(\log 1/\delta)$ copies of the estimate, we must update wd entries in the sketch for every update. For small values of ϵ this can be a large overhead, and means that such an approach does not easily scale to very

high speed data stream environment without special purpose hardware. However, a simple hashing trick has been applied to increase the speed significantly [6–8, 22]. Instead of keeping w copies of the estimate and taking the average of their estimates, the key idea is to use a second hash function f which maps each item i onto $[1..w]$, and only updating the estimate $f(i)$. To produce an estimate of the join or self-join size, the sum of the estimates is used instead of the average. Mathematically this gives the same expectation and variance as the original method, but requires only $O(d)$ estimates to be modified for each update instead of $O(wd)$. This means that the cost is essentially independent of the accuracy parameter ϵ , and depends only on $\log 1/\delta$, which is typically quite small in practice.

4.3 Multidimensional and Spatial Data

We have so far phrased the discussion in terms of join sizes and large vectors. However, one can model other structures with sketches. By appropriate linearization, one can make a sketch of a large matrix of values, and similar higher dimensional structures. Such modeling is necessary in order to estimate the size of multi-way joins (see the chapter of Dobra et al. in this volume).

More generally, one can also apply the sketching technique to summarize spatial data. Now, the input consists of (a stream of) objects in low dimensional space (say, two or three dimensions). The notion of a spatial join is to count the number of object pairs that fall within a certain (specified as part of the query) distance of each other. Other natural queries are to ask for the approximate number of points falling within a given range, e.g., a rectangular or cuboid region.

Sketch data structures have been applied to these problems [10]. The key technique is to build sketches from the query specifications in such a way that the approximate join size between the query sketch and data sketch is an unbiased estimator for the answer to the query. For example, suppose the input consists of a set of one-dimensional ranges $[a \dots b]$. To answer the query of how many intervals contain a given point c , we can simply pose the point query I_c . To count how many ranges overlap with a query range, we can compute the sum of how many of the ranges contain each end point of the query range, and how many of the end points of the ranges are contained within the query range. Assuming that the query end points do not coincide with points of the input,² this query returns exactly twice the number of intersections, and so can be approximated correctly in expectation.

In order to bound the error from using sketches, and ensure that updates are fast to process, ranges are not represented directly, but using the “dyadic range decomposition” described in Sect. 4.1. This approach can be generalized from one dimensional data to points and rectangles in the plane and in three dimensional space, etc. This shows some nice features of the sketching approach: it is sufficiently general that it

²This assumption can be removed with some careful manipulation; see [10].

can be applied to a variety of different streaming scenarios. Using techniques such as the dyadic range decomposition, these ideas can be implemented efficiently and with bounded error per update.

4.4 Further Extensions and Applications

We outline some of the many other applications in data streams that sketch-based techniques have found:

- **Vector L_1 Difference.** For some applications, rather than the L_2 distance between two vectors, $\sqrt{F_2(\mathbf{x} - \mathbf{y})}$, it is necessary to compute the L_1 difference, $\sum_{i=1}^U |\mathbf{x}[i] - \mathbf{y}[i]|$. This can be reduced to F_2 by representing the vectors in unary notation, since $L_1(\mathbf{x} - \mathbf{y}) = F_2(\mathbf{x} - \mathbf{y})$ if \mathbf{x} and \mathbf{y} are binary (zero/one) vectors. However, in order to compute this transformation efficiently, new methods are needed to quickly compute the sum of 4-wise hash functions, rather than explicitly creating the unary representation of large vector entries [11].
- **Triangle Counting in Graphs.** Many data streams represent graphs, presented as streams of edges, and the streaming challenge is to compute properties of the induced graphs. The number of triangles, also known as the clustering coefficient, occurs in a variety of applications, but seems challenging to compute when the edges forming each triangle can be arbitrarily interleaved in the stream. However, by a careful transformation, the number of triangles can be expressed as a function of appropriately defined frequency moments F_0 , F_1 and F_2 [4]. New techniques are required to efficiently update the sketches as each edge requires a large number of updates to be applied, but these updates can be described concisely as a range of values.
- **Change Detection.** In many large scale monitoring applications, the fundamental question is whether the current observations are in line with predicted behavior, or whether they appear to be at odds with what was expected. Such applications are generally known as “change detection”. A general approach was suggested in [17]: build sketches of recent data, and then apply various standard modeling techniques to combine these into a sketch of the predicted behavior. The new data can then be observed, and tested against the prediction: either in terms of individual item counts or by comparing the whole stream. This approach relies crucially on the linearity properties of the sketch transformation, so the predicted sketch can be obtained by applying the prediction model to the historical sketches. Thus, the whole method can be carried out in small space and at high speeds in the streaming model.
- **Quantiles under insertions and deletions.** The problem of tracking quantiles over a stream of input items drawn from $[1..U]$ is, given ϕ , return an item whose rank is (approximately) ϕN . Various techniques have been proposed for this problem when the stream consists of insertions of items only. However, when the input stream may also contain deletions of items that have previously appeared, these techniques do not apply. Observe that the quantiles query can be restated

as a range query: if we can estimate how many items fall in the range $[1..R]$, then we can binary search for the value of R whose range contains ϕN points. This range query can be answered using the techniques of Sect. 4.1. The reduction to dyadic ranges makes updates and queries reasonably fast, and the error is bounded by $\epsilon \log U \sqrt{F_2(x)}$. Because deletions can be processed as negative updates to sketches, it is easy to see that deletion operations are handled correctly. Full details of this approach to finding quantiles using sketches are in [7, 13].

- **Tracking Queries over Distributed Streams.** Large-scale stream processing applications rely on continuous, event-driven monitoring, and are often inherently *distributed*, with several remote monitor sites observing their local, high-speed data streams and exchanging information through a communication network. This distribution of the data naturally implies critical *communication constraints* that typically prohibit continuously centralizing all the streaming updates, due to volume and speed of the streams that can easily overwhelm the underlying network. Monitoring queries over such *distributed streams* raises a host of new challenges. A property of AMS sketches that makes them particularly interesting in this setting is that, due to their linear nature, they are naturally *composable* through simple vector addition. In other words, given two “parallel” AMS sketches (built using the same 4-wise hash functions) over two different streams, the sketch of the combined stream (i.e., the union of the two streams) is simply the component-wise summation of their sketches. More details on the distributed streaming model and results can be found in a chapter by Garofalakis later in this volume.

5 Concluding Remarks

The original paper describing the sketching technique discussed here was published in 1996 [3], and showed the F_2 application. A subsequent paper in 1999 [2] extended the results to join and self-join size. The original “AMS” paper considers a broad range of problems based on the frequency moments, and has come to be viewed as one of the foundational works on data streams, even though this term is not explicitly used by the authors. In addition to the results on F_2 , the authors also give space efficient algorithms for all frequency moments F_k , $k \in \mathcal{N}$, and lower bounds for the problems showing that, for $k \geq 6$, space polynomial in n is required. This led to a sequence of papers in the theoretical computer science computer science which has focussed on improving the upper and lower bounds for the frequency moments problem for $k \geq 3$, culminating in recent results showing essentially tight upper and lower bounds for these problems.

The result can also be thought of in terms of embedding vectors into lower dimensional spaces. The Johnson–Lindenstrauss lemma [16] proved that there exists embeddings of vectors in Euclidean space into a Euclidean space of (smaller) dimension $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ which preserves distances up to a $(1 \pm \epsilon)$ factor. We can view the sketch technique here as an explicit embedding into a Euclidean-like space (the

operations of averaging and median finding mean that we cannot treat the sketches as vectors in Euclidean space), which is computable in a data stream setting with small space and limited (four-wise) randomness. The results of Achlioptas [1] show that, if we assume full randomness, then we can also operate in Euclidean space.

Lastly, several efficient implementations of sketch data structures have been made and published on the Internet (e.g., <http://www.cs.rutgers.edu/~muthu/massdall-code-index.html>). These can be freely modified and used as the basis of more complex data stream algorithms.

References

1. D. Achlioptas, Database-friendly random projections, in *Proceedings of ACM Principles of Database Systems* (2001), pp. 274–281
2. N. Alon, P. Gibbons, Y. Matias, M. Szegedy, Tracking join and self-join sizes in limited storage, in *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems* (1999), pp. 10–20
3. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*. (1996), pp. 20–29. Journal version in: J. Comput. Syst. Sci. **58**, 137–147 (1999)
4. Z. Bar-Yossef, R. Kumar, D. Sivakumar, Reductions in streaming algorithms, with an application to counting triangles in graphs, in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (2002), pp. 623–632
5. J.L. Carter, M.N. Wegman, Universal classes of hash functions. J. Comput. Syst. Sci. **18**(2), 143–154 (1979)
6. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)* (2002), pp. 693–703
7. G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, in *Latin American Informatics* (2004), pp. 29–38
8. G. Cormode, M. Garofalakis, Approximate continuous querying over distributed streams. ACM Trans. Database Syst. **33**(2) (2008)
9. G. Cormode, M. Garofalakis, P.J. Haas, C. Jermaine, Synopses for massive data: samples, histograms, wavelets, sketches. Found. Trends® Databases **4**(1–3) (2012)
10. A. Das, J. Gehrke, M. Riedewald, Approximation techniques for spatial data, in *Proc. of the 2004 ACM SIGMOD Intl. Conference on Management of Data* (2004)
11. J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, An approximate L_1 -difference algorithm for massive data streams, in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (1999), pp. 501–511
12. M. Garofalakis, P.B. Gibbons, Approximate query processing: taming the terabytes, in *27th Intl. Conf. on Very Large Data Bases*, Rome, Italy (2001). Tutorial
13. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, How to summarize the universe: dynamic maintenance of quantiles, in *Proceedings of the International Conference on Very Large Data Bases* (2002), pp. 454–465
14. I.J. Good, Surprise indexes and p -values. J. Stat. Comput. Simul. **32**, 90–92 (1989)
15. Y.E. Ioannidis, S. Christodoulakis, Optimal histograms for limiting worst-case error propagation in the size of join results. ACM Trans. Database Syst. **18**(4), 709–748 (1993)
16. W.B. Johnson, J. Lindenstrauss, Extensions of Lipschitz mapping into Hilbert space. Contemp. Math. **26**, 189–206 (1984)
17. B. Krishnamurthy, S. Sen, Y. Zhang, Y. Chen, Sketch-based change detection: methods, evaluation and applications, in *Proceedings of the ACM SIGCOMM Conference on Internet Measurement* (2003), pp. 234–247

18. E. Kushilevitz, N. Nisan, *Communication Complexity* (Cambridge University Press, Cambridge, 1997)
19. R. Motwani, P. Raghavan, *Randomized Algorithms* (Cambridge University Press, Cambridge, 1995)
20. S. Muthukrishnan, Data streams: algorithms and applications, in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms* (2003)
21. M. Thorup, Even strongly universal hashing is pretty fast, in *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (2000), pp. 496–497
22. M. Thorup, Y. Zhang, Tabulation based 4-universal hashing with applications to second moment estimation, in *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (2004), pp. 615–624
23. M.N. Wegman, J.L. Carter, New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.* **22**(3), 265–279 (1981)

Top- k Frequent Item Maintenance over Streams

Moses Charikar

1 Introduction

In this chapter, we consider the problem of finding the most frequent items in a data stream. Given a data stream a_1, a_2, \dots, a_n , where each $a_i \in \{1, \dots, m\}$, we would like to identify the items that occur most frequently in one pass over the data stream using a small amount of storage space. Such problems arise in a variety of settings. For example, a search engine might be interested in gathering statistics about its query stream and in particular, identifying the most popular queries. Another application is to detecting network anomalies by monitoring network traffic.

Throughout this chapter, for ease of exposition, we will assume that the elements of the data stream are integers in $[1, m]$. In general, the items in the data stream could be more complex objects (e.g., queries to a search engine). However, such complex objects can always be mapped to integers by hashing or other means. Hence without loss of generality, we may assume that the data stream elements are in fact integers.

The brute-force approach to finding frequent elements would involve maintaining counters for every distinct element seen so far. However, this requires far too much storage—linear in the number of distinct elements. In fact, if we insist on determining the most frequent element exactly, using linear storage is unavoidable. Think of the problem of determining the most frequent element in a data stream where all elements are distinct except for one. Determining the one element that repeats requires linear storage. We will prove this formally later. However, if we allow the algorithm some slack and accept some error in the results produced, then ingenious solutions that require limited space are possible. We will discuss several algorithmic approaches developed to solve this problem. In addition to allowing some errors in the output, we will also consider randomized algorithms. Such algorithms will also

M. Charikar (✉)

Computer Science Department, Stanford University, Stanford, CA 94305, USA
e-mail: moses@cs.stanford.edu

have a small failure probability δ , which can usually be made arbitrarily by increasing the storage space. We will guarantee that the randomized produces an output within the guaranteed error bounds with probability $1 - \delta$.

First we introduce some notation we will use throughout this chapter. For item i , let m_i denote the number of occurrences of item i in the data stream. Let n be the total size of the data stream. $n = \sum_i m_i$. We define the frequency of item i to be the fraction m_i/n .

We consider different variants of the frequent items problem:

1. **Hot items.** Given an integer k , we would like to identify all items that occupy more than $1/k$ fraction of the data stream. In general, the algorithms we describe for this problem return all hot items, but also return certain items that are not hot. These could be eliminated in a second pass over the data, or it might be the case that such false positives do not affect the application adversely.
2. **Top k items with error.** Given an error bound, we would like to determine the top k items with a small error in the item counts. The goal of the algorithm is to return a subset F of k items such that for any pair of items $i \in F, j \notin F, m_j - m_i$ is less than a certain error bound. Here the error bound might be of the form εN , or may be a more complicated function of the distribution of the m_i 's.

The algorithms we describe to solve such problems will typically maintain data structures which can be used to estimate the item counts with small error. These estimated counts can be used to maintain the approximate top k items in one pass over the data stream.

One desirable property that some of the algorithms we describe have is that they can handle item insertions as well as deletions. In our exposition, we mainly focus on the insertions only setting, but we will point out possible extensions to handle deletions.

We describe a variety of different approaches that have been proposed to solve these problems. Our goal is to give a flavor of the various techniques that have been used in this area. In Sect. 2, we describe random sampling based approaches. In Sect. 3, we look at some deterministic approaches. In Sect. 4, we describe approaches that maintain approximate frequency counts for all items by maintaining multiple counters. Finally, in Sect. 5, we show lower bounds on the space requirements for streaming algorithm that identify frequent items.

2 Random Sampling Methods

Given a random sample of size $O(1/\varepsilon^2 \log(1/\delta))$, one can determine item frequencies up to an additive error of ε fraction of the total count of all items with probability $1 - \delta$. Consider a random sample of size $S = O(1/\varepsilon^2 \log(1/\delta))$. For item i , let \hat{p}_i denote the fraction of the sample occupied by item i . Let p_i denote $\mathbf{E}[\hat{p}_i]$. Then $p_i = m_i/n$. Then $|\hat{p}_i - p_i| \leq \varepsilon$ with probability $1 - \delta$. This is an easy consequence of Chernoff bounds. In fact, a random sample of the data stream can be maintained

in one pass over the data using reservoir sampling. So this gives a simple randomized method to identify frequent items. In the following sections, we will improve on this basic technique.

2.1 Sticky Sampling

Manku and Motwani [8] proposed a method called *sticky sampling* to identify frequent items. This improves on the space requirements of the basic random sampling approach. Their method guarantees that with high probability all items with counts at least n/k are identified correctly, no items with counts less than $(1/k - \varepsilon)n$ are reported, and further all estimated frequencies are less than the true frequencies by at most εn . If an algorithm has this guarantee on its output, it is said to maintain an ε -deficient synopsis.

We give a brief overview of the ideas behind Sticky Sampling, but do not go into details since the performance of this algorithm is superseded by the deterministic methods we will describe next. The idea behind their approach is to maintain a random sample of the data stream, where the sampling rate is adapted to the length of the stream seen so far. Sampling at rate r refers to placing elements in the random sample with probability $1/r$. Multiple copies of the same item in the sample are represented using a counter for the item. Let $t = \frac{1}{\varepsilon} \log(k/\delta)$. The first $2t$ elements are sampled at rate $r = 1$, then next $2t$ are sampled at rate $r = 2$, the next $4t$ elements are sampled at rate $r = 4$, and so on. As the sampling rate is changed, the old sample is updated to reflect the new sampling rate. This may result in reducing item counter values—if reduced to 0, the corresponding item is removed from the sample.

Manku and Motwani show that the space requirement for this algorithm is $\frac{2}{\varepsilon} \log(k/\delta)$ in expectation and it computes an ε -deficient synopsis with probability $1 - \delta$.

3 Deterministic Methods

Misra and Gries [9] proposed a method to find all items of frequency greater than $1/k$ in one pass. This is a generalization of a well known linear time algorithm to find a majority element in an array. It is interesting to note that Misra and Gries studied this problem and proposed this algorithm several years before the notion of streaming algorithms was developed. Much later, their result was rediscovered by two sets of authors: Karp, Papadimitriou and Shenker [7] and Demaine, López-Ortiz and Munro [5], who both also gave a more efficient implementation than the one proposed by Misra and Gries.

The idea behind the algorithm is to maintain a table of elements and associated counters and update this as the stream is read. Every element occurs at most once in the table and at any point of time, there are at most k distinct elements in the table.

In the end, all elements with frequency strictly larger than $1/k$ are guaranteed to be present in the table. Additional elements may be present as well. These could be identified and eliminated by performing a second pass over the data.

Initially, the algorithm starts with an empty table T . When a new element x of the stream is read, we first check to see if x is present in the table T . If so, we increment its counter by 1. If not, we add the element counter pair $(x, 1)$ to T . Finally, if there are k distinct elements in T , we subtract 1 from each of their associated counters and any elements with counter value 0 are eliminated from the table.

Claim Any element with frequency strictly larger than $1/k$ will be present in the table T at the end of the data stream.

Proof We can view the algorithm as maintaining a multiset of elements as follows. Every element of the data stream is added to the current multiset. If there are k distinct elements present currently, one copy of each of the k distinct elements is removed from the multiset. Suppose x is an element which occurs strictly more than n/k times in the data stream. For each of these occurrences of x , a copy of x is added to the multiset maintained by the algorithm and some of these copies might be deleted later. We claim that at most n/k copies are deleted. Note that whenever x is deleted from the multiset, one copy is deleted at a time and this is accompanied by the deletion of $k - 1$ elements other than x . Since the stream has n elements in all, at most n/k copies of x can be deleted. Thus, at least one copy of x survives in the multiset maintained by the algorithm. \square

We mention that this algorithm can easily be adapted to produce an ε -deficient synopsis (mentioned in the previous section) using a table size of $1/\varepsilon$. Any item i that occurs more than $m_i = f_i \times n$ times in the data stream will have at least $(f_i - \varepsilon)n$ copies in the table T constructed thus. The modified algorithm reports all items with more than $(1/k - \varepsilon)n$ copies in T . This includes items with actual counts greater than n/k , and no items with count less than $(1/k - \varepsilon)n$.

We also mention that another deterministic approach called Lossy Counting was proposed by Manku and Motwani [8]. In fact the algorithm is similar in several aspects to another algorithm proposed by Misra and Gries [9] for identifying items of frequency greater than $1/k$. However, Misra and Gries were unable to analyze the space requirements of this algorithm, while Manku and Motwani showed that their Lossy Counting algorithm requires space $k \log(N/k)$. Although the worst case space requirements of this algorithm are worse than the deterministic algorithm presented earlier, Manku and Motwani point out that Lossy Counting may perform better in practice depending on the distribution of items in the data stream. We omit the details of the Lossy Counting algorithm and refer the interested reader to [8].

4 Randomized Multiple Counter Methods

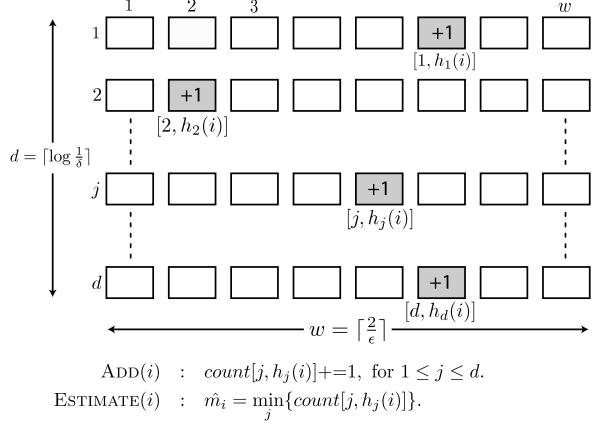
Several methods have been proposed where the algorithm maintains a collection of counters incrementally as it scans the data stream. Every element of the data stream

causes an update to several counters. Collectively, the set of counters can be used to estimate the number of times a data item has been seen so far. In other words, the algorithm maintains a data structure that estimates the number of occurrences so far for any data item. These estimates have additive errors which can be bounded as a function of the characteristics of the data stream. Ideally, if we had exact counts as we scanned the data stream, this could be used to maintain an exact list of the k most frequent items in the data stream. Similarly, the noisy estimates from the approximate counting data structure can be used to maintain an approximate list of the most frequent elements.

The guarantee provided by such an algorithm is that the frequency of any of the k elements reported by the algorithm is at least the frequency of the k th most frequent element minus the error guarantee of the count data structure. In addition to the additive errors, the data structures also have a failure probability δ for a single query, i.e., with probability δ , we cannot say anything meaningful about the estimate produced. In using such a approximate counting data structure for estimating frequent items, we will need to query the data structure for every element of the data stream. In order to be conservative, if any of these queries fails, we will say that the frequent item estimation has failed. If we want failure probability δ' overall, we set the failure probability for an individual query to be $\delta = \delta'/n$. By the union bound, the probability that any one of the n queries to the data structure fails is at most $n\delta = \delta'$. In the subsequent analysis, the time and space requirements of the data structures is expressed in terms of the failure probability δ of an individual query. Typically, the dependence on δ is of the form $\log(1/\delta)$. In order to translate this into a failure probability δ' for the overall frequent items computation, we need to substitute $\delta = \delta'/n$. This usually gives a dependence of the form $\log(n/\delta') = (\log n + \log(1/\delta'))$.

The first two data structures we present are the Count-Min sketch and the Count Sketch. The Count-Min sketch was actually developed after the Count Sketch, but is simpler to explain and analyze. It also has some common features with the Count Sketch, so we present it first. All the algorithms we present have the property that they maintain synopsis data structures consisting of a collection of counters that are additive. Thus element deletions can be handled by performing the reverse operation to that performed when adding an element—decreasing a counter instead of increasing it. While the first two data structures (the Count-Min sketch and the Count Sketch) permit updates to the data structure to reflect deletions, the data structure itself does not contain information that allows the identification of frequent items. In other words, approximate counts can be obtained, but the determination of frequent items requires some additional information to be stored. In one pass over the data, frequent items can be identified while performing updates to the data structure in an insertions only scenario. This can be done by maintaining a heap containing the top k elements seen so far and their estimated counts from the synopsis data structure. When a new element of the stream is encountered, the synopsis data structure is updated. If the new element is already in the heap, its count is incremented. Otherwise, we obtain an estimate of its count from the synopsis data structure. If this estimate exceeds the smallest estimated count in the heap, the new element is added to the heap and the element with the smallest count is evicted.

Fig. 1 The array of counters used by the Count-Min sketch data structure, and its maintenance algorithms. Shaded cells indicate the counter values incremented for an item i



With both inserts and deletes, a separate pass would be required to actually identify the frequent items. The last approach we describe produces a data structure that can be updated with insertions and deletions of elements, but also allows identification of candidate frequent elements.

4.1 Count-Min Sketch

The Count-Min Sketch was introduced by Cormode and Muthukrishnan [4]. The data structure (see Fig. 1) consists of a two-dimensional array of counters with width w and depth d , i.e., there are $w \times d$ counters $count[i, j]$ for $1 \leq i \leq w$ and $1 \leq j \leq d$. Each of the counters is initially set to zero. The data structure takes two parameters ϵ and δ that control the error guarantee, $w = \lceil \frac{2}{\epsilon} \rceil$ and $d = \lceil \log \frac{1}{\delta} \rceil$. In addition, d hash functions $h_1, \dots, h_d : \{1, \dots, m\} \rightarrow \{1, \dots, w\}$ are chosen uniformly at random from a pairwise independent family.

When a new item i is seen, one counter in each row is incremented, where the counter in row j is determined by h_j . In other words, $count[j, h_j(i)]$ is incremented by 1. At any point of time, the estimated count for item i is given by $\hat{m}_i = \min_j count[j, h_j(i)]$. The following theorem gives guarantees on the estimates \hat{m}_i .

Theorem 1 Let m_i be the number of times item i occurs in the data stream and let n denote the total number of items seen so far. Then, $m_i \leq \hat{m}_i$ and with probability at least $1 - \delta$, $\hat{m}_i \leq m_i + \epsilon n$.

Proof Consider the indicator variable $I_{i,j,k}$ which is 1 iff $i \neq k$ and $h_j(i) = h_j(k)$, and 0 otherwise. By the pairwise independence of hash functions h_j ,

$$\mathbf{E}[I_{i,j,k}] = \Pr[h_j(i) = h_j(k)] \leq \frac{\epsilon}{2}.$$

The last inequality follows from the fact that the range of h_j is has size at least $2/\varepsilon$. Define random variable $X_{i,j}$ to be $X_{i,j} = \sum_{k=1}^m I_{i,j,k} m_k$. Note that $\text{count}[j, h_j(i)] = a_i + X_{i,j}$. Clearly, $\hat{m}_i = \min_j \text{count}[j, h_j(i)] \geq m_i$. For the other direction, note that

$$\mathbf{E}[X_{i,j}] = \mathbf{E}\left(\sum_{k=1}^m I_{i,j,k} m_k\right) \leq \sum_{k=1}^m m_k \mathbf{E}[I_{i,j,k}] \leq \frac{\varepsilon}{2} n$$

by linearity of expectation. By the Markov inequality,

$$\begin{aligned} \Pr[\hat{m}_i > m_i + \varepsilon n] &= \Pr[\forall j \text{count}[j, h_j(i)] > m_i + \varepsilon n] \\ &= \Pr[\forall j m_i + X_{i,j} > m_i + \varepsilon n] \\ &= \Pr[\forall j X_{i,j} > 2\mathbf{E}[X_{i,j}]] < 2^{-d} \leq \delta. \end{aligned} \quad \square$$

4.2 Count Sketch

The Count Sketch was introduced by Charikar, Chen and Farach-Colton [2]. These techniques are particularly suited to heavy-tailed distributions and significantly outperform other methods discussed for such distributions, e.g., for Zipfian distributions with exponent less than 1. Before we go into the details of the data structure and its analysis, we give a simpler scheme for estimating the most frequent element. This will serve as a motivation for the full construction later.

Estimating the Most Frequent Element

Consider the following scheme for estimating the count of the most frequent element. We maintain a single counter c that is initially zero. We pick a hash function $s : \{1, \dots, m\} \rightarrow \{+1, -1\}$ from a pairwise independent family. When item i is seen, we add $s(i)$ to the counter. At any point of time, the counter value is $\sum_{i=1}^m m_i s(i)$, where m_i is the number of occurrences of item i so far. The estimate \hat{m}_i of the count of i is simply $s(i) \times c$.

Lemma 1 $\mathbf{E}[\hat{m}_i] = m_i$ and $\mathbf{Var}[\hat{m}_i] = (\sum_{j=1}^m (m_j)^2) - (m_i)^2$.

Proof Note that $E[s(i)s(j)] = 0$ for $i \neq j$ by the pairwise independence of s . This is the crucial fact underlying the analysis.

$$\begin{aligned} \hat{m}_i &= s(i) \sum_{j=1}^m m_j s(j), \\ \mathbf{E}[\hat{m}_i] &= \sum_{j=1}^m m_j \mathbf{E}[s(i)s(j)] = m_i. \end{aligned}$$

The last equality follows from the pairwise independence of s . Now, we analyze the variance of \hat{m}_i :

$$\begin{aligned}
 (\hat{m}_i)^2 &= \left(s(i) \sum_{j=1}^m m_j s(j) \right)^2, \\
 \mathbf{E}[(\hat{m}_i)^2] &= \sum_{j,k} m_j m_k \mathbf{E}[s(j)s(k)] \\
 &= \sum_j (m_j)^2 + \sum_{j \neq k} m_j m_k \mathbf{E}[s(j)s(k)] = \sum_j (m_j)^2, \\
 \mathbf{Var}[\hat{m}_i] &= \mathbf{E}[(\hat{m}_i)^2] - (\mathbf{E}[\hat{m}_i])^2 = \sum_j (m_j)^2 - m_i^2. \quad \square
 \end{aligned}$$

In general, the variance of this estimator may be too high. This is easily addressed by taking multiple independent copies of the estimator and averaging them. Suppose we take d independent copies of this estimator and return the average of their values. Then the following lemma shows that the estimate obtained is tightly concentrated around the mean, where the concentration can be made tighter by increasing d .

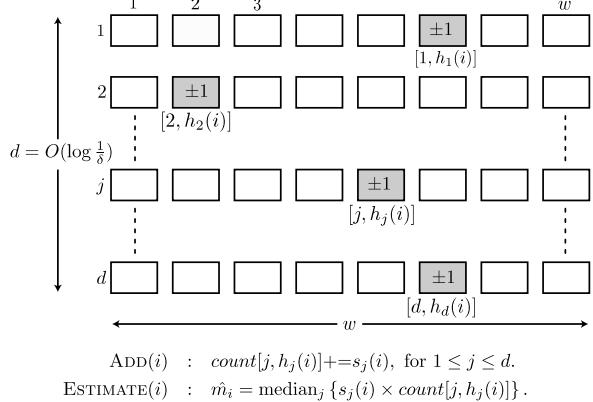
Lemma 2 *Let \hat{m}_i^d be the estimator obtained by averaging d independent copies of the estimator above. If the variance of each individual estimator is V , then $\mathbf{E}[\hat{m}_i^d] = m_i$, $\mathbf{Var}[\hat{m}_i^d] = V/d$ and further, with probability $1 - \delta$, $|\hat{m}_i^d - m_i| < \sqrt{(V/d) \log(1/\delta)}$.*

Notice that the error guarantee depends on the sum of the squares of the counts of the items. This makes this technique especially attractive for heavy tailed distributions where the most frequent element occupies a relatively small fraction of the data stream. Consider, for example, a Zipfian distribution with exponent $1/2$. In this case, the most frequent element occupies an $O(1/\sqrt{m})$ fraction of the data stream. In order to detect this, the other methods we discussed would require $\Omega(\sqrt{m})$ space. However, the method discussed would identify the most frequent element with high probability using only $O(\log m)$ space.

Estimating the Top k Elements

We now adapt the ideas presented in the previous section to identifying the top k elements. The main obstacle in doing this is that the error of the estimator presented previously might be too high for identifying the lowest frequency element we are interested in—the k th most frequent element. The problem is that the variance is affected by the sum of squares of the counts of the top k items. To address this issue, we replace each counter by a row of counters, producing a two-dimensional array of counters. Each item will now update exactly one counter in every row, chosen by a hash function. By doing this, we ensure that high frequency elements typically update different counters in a row. Consider the k th high frequency element that was

Fig. 2 The array of counters used by the Count Sketch data structure, and its maintenance algorithms. Shaded cells indicate the counter values modified for an item i



potentially problematic before. The counter it is assigned to in every row is typically not affected by any other high frequency element, i.e., none of the other top k items are assigned to the same counter. Thus, the variance associated with this counter is usually not high. Note that we cannot guarantee this, but we can pick the number of counters in each row to be large enough so that this happens with high probability. However, there is still a small probability that the counter in a row is tainted by other high frequency elements. If we average counter values in each row as before, a small number of tainted counters could affect the final estimate adversely. To get around this problem, we take the median of the counters assigned to an element in every row. As long as less than half the rows have tainted counters, the median estimator will be good.

The data structure (see Fig. 2) consists of a two-dimensional array of counters with width w and depth d , i.e., there are $w \times d$ counters $\text{count}[i, j]$ for $1 \leq i \leq w$ and $1 \leq j \leq d$. $d = O(\log(1/\delta))$ where δ is a parameter that specifies the desired error probability. w controls the additive error in the frequency estimates, however the dependence is somewhat complex. w must be set large enough to make the additive error acceptably small. Each of the counters is initially set to zero. d hash functions $h_1, \dots, h_d : \{1, \dots, m\} \rightarrow \{1, \dots, w\}$ are chosen uniformly at random from a pairwise independent family. In addition, d hash functions $s_1, \dots, s_d : \{1, \dots, m\} \rightarrow \{+1, -1\}$ are chosen uniformly at random from a pairwise independent family.

When a new item i is seen, one counter in each row is incremented or decremented, where the counter in row j is determined by h_j . The value $s_j(i)$ is added to counter $\text{count}[j, h_j(i)]$. At any point of time, the estimated count for item i is computed as follows. The estimated count of item i for row j is given by $\hat{m}_i(j) = s_j(i) \times \text{count}[j, h_j(i)]$. The overall estimates count for item i is given by $\hat{m}_i = \text{median}_j \hat{m}_i(j)$. It will be convenient to express the error guarantee in terms of the following parameter γ ,

$$\gamma = \sqrt{\frac{\sum_{i>w/8} (m_i)^2}{w}}. \quad (1)$$

The following theorem gives guarantees on the estimates \hat{m}_i .

Theorem 2 *With probability $1 - \delta$, $|\hat{m}_i - m_i| \leq 8\gamma$.*

Proof We will first prove that with constant probability, the counter in any row provides a good estimate of m_i . Then, we will use this to show that the median of counter values over all rows is a good estimator.

Let $A_j(i)$ be the set of items that map to the same counter that item i maps to in the j th row if the data structure, i.e., $A_j(i) = \{i' : i' \neq i, h_j(i') = h_j(i)\}$. Let $A_j^>(i)$ be the elements of $A_j(i)$ other than the $w/8$ most frequent elements. Let $v_j(i) = \sum_{i' \in A_j(i)} (m_{i'})^2$. $v_j^>(i)$ is defined analogously for $A_j^>(i)$.

We first claim that $\mathbf{E}[\hat{m}_i(j)] = m_i$ and $\mathbf{Var}[\hat{m}_i(j)] = v_j(i)$:

$$\begin{aligned} \text{count}[j, h_j(i)] &= m_i s_j(i) + \sum_{i' \in A_j(i)} m_{i'} s_j(i'), \\ \hat{m}_i(j) &= s_j(i) \times \text{count}[j, h_j(i)] = m_i + \sum_{i' \in A_j(i)} m_{i'} s_j(i') s_j(i), \\ \mathbf{E}[\hat{m}_i(j)] &= m_i + \sum_{i' \in A_j(i)} m_{i'} \mathbf{E}[s_j(i') s_j(i)] = m_i, \\ \mathbf{Var}[\hat{m}_i(j)] &= \mathbf{E}[(\hat{m}_i(j) - m_i)^2] = \mathbf{E}\left[\sum_{i_1, i_2 \in A_j(i)} m_{i_1} m_{i_2} s_j(i_1) s_j(i_2)\right] \\ &= \sum_{i' \in A_j(i)} (m_{i'})^2 + \sum_{i_1, i_2 \in A_j(i), i_1 \neq i_2} m_{i_1} m_{i_2} \mathbf{E}[s_j(i_1) s_j(i_2)] \\ &= \sum_{i' \in A_j(i)} (m_{i'})^2 = v_j(i). \end{aligned}$$

Further, we claim that

$$\mathbf{E}[v_j^>(i)] = \frac{\sum_{i' > w/8} (m_{i'})^2}{w}.$$

Note that

$$\begin{aligned} v_j^>(i) &= \sum_{i' \in A_j(i), i' > w/8} (m_{i'})^2, \\ \mathbf{E}[v_j^>(i)] &= \sum_{i' > w/8, i' \neq i} (m_{i'})^2 \mathbf{Pr}[i' \in A_j(i)] \\ &\leq \sum_{i' > w/8} (m_{i'})^2 \times (1/w), \end{aligned}$$

where the last inequality follows from the pairwise independence of h_j .

We will say that $\hat{m}_i(j)$ is good if $|\hat{m}_i(j) - m_i| \leq 8\gamma$. Now we show that this happens with probability strictly greater than $1/2$. To do this, we define three bad events

and show that each of them occurs with small probability. If none of them occur, then the estimator $\hat{m}_i(j)$ is good. Since the probability that any element i' is present in $A_j(i)$ is $1/w$ (by pairwise independence of h_j), the probability that $A_j(i)$ contains any of the top $w/8$ elements is at most $1/8$ (by the union bound). Call this event $\text{COLLISIONS}_j(i)$. Note that if $\text{COLLISIONS}_j(i)$ does not occur, then $A_j^>(i) = A_j(i)$ and $v_j^>(i) = v_j(i)$. Since $\mathbf{E}[v_j^>(i)] = \frac{\sum_{i'>w/8}(m_{i'})^2}{w}$, the probability that $v_j^>(i)$ is greater than $8\frac{\sum_{i'>w/8}(m_{i'})^2}{w}$ is less than $1/8$ by Markov's inequality. Call this event $\text{HIGH-VARIANCE}_j(i)$. Note that if $\text{COLLISIONS}_j(i)$ and $\text{HIGH-VARIANCE}_j(i)$ both do not occur then $\mathbf{Var}[\hat{m}_i(j)] = v_j(i) = v_j^>(i) \leq 8\frac{\sum_{i'>w/8}(m_{i'})^2}{w}$. By Chebyshev's inequality,

$$\Pr[|\hat{m}_i(j) - m_i| > \sqrt{8 \mathbf{Var}[\hat{m}_i(j)]}] < 1/8.$$

Call this last event $\text{LARGE-DEVIATION}_j(i)$. The probability of each of the three bad events we defined is at most $1/8$. By the union bound, with probability at least $1 - 1/8 - 1/8 - 1/8 = 5/8$, none of the events $\text{COLLISIONS}_j(i)$, $\text{HIGH-VARIANCE}_j(i)$, $\text{LARGE-DEVIATION}_j(i)$ occur. In this case,

$$|\hat{m}_i(j) - m_i| \leq \sqrt{64 \frac{\sum_{i'>w/8}(m_{i'})^2}{w}} = 8\gamma.$$

To finish the proof, we need to show that $|\hat{m}_i - m_i| \leq 8\gamma$ with probability at least $1 - \delta$. Recall that $\hat{m}_i = \text{median}_j \hat{m}_i(j)$. If an index $j \in [1, d]$ satisfies the property that $\hat{m}_i(j)$ is good, then we call j a good index. If we can show that more than $d/2$ indices $j \in [1, d]$, are good with high probability, this would imply the required error bound for \hat{m}_i . This follows easily from Chernoff bounds, since the expected number of good indices is $5d/8$ and $d = \Omega(\log(1/\delta))$. \square

4.3 Group-Testing Approaches

Cormode and Muthukrishnan [3] gave an algorithm for identifying frequent items based on group-testing ideas. The goal is to identify all items which occur with frequency larger than $1/k$. The techniques presented here are applicable to data streams with both insert and delete operations.

To motivate the technique, we first describe a scheme to find a majority element, i.e., an element with frequency more than $1/2$. Consider the elements $1, \dots, m$ represented in binary. Let $r = \lceil \log_2 m \rceil$ for each bit position $j \in \{1, \dots, r\}$, we consider the subset of items S_j consisting of all elements that have a 1 in bit position j . Also define S_0 to be the set of all items. We maintain $r + 1$ counters C_0, \dots, C_r , initially zero, where C_j is the total count of all elements in set S_j . Clearly these counters can be maintained incrementally in one scan of the data stream, by incrementing appropriate counters when an element is added and decrementing counters when an

element is deleted. (We will assume that the total number of deletions of an element never exceed the number of insertions.)

How do we identify a majority element using these counters? Suppose there is a majority element x . Each bit of the binary representation of this element can be deduced from the set of counters maintained. Note that C_0 is the total count of all elements in S_0 , i.e., all the elements. Consider the j th bit of the majority element x . If this bit is 1, then $x \in S_j$. Then the value of C_j exceeds $C_0/2$. If this bit is 0, then $x \notin S_j$. Then the value of C_j is less than $C_0/2$. Thus comparing the value C_j to $C_0/2$ enables us to identify the j th bit of the majority element. Note that this scheme operates correctly if there is indeed a majority element. If no such element exists, it will still report a majority element, i.e., it comes up with false positives.

Now we adapt this scheme to identify elements whose frequency exceeds $1/k$. Note that there are fewer than k such elements—call them hot items. The main idea is to divide the elements into a small number of (overlapping) subsets and run a modification of the majority identification scheme for each subset. The collection of subsets is constructed in such a way that for every hot item x , there is a subset for which x is the majority element. To achieve this goal, for every hot item x , we ensure that there is a *good* set S containing x such that the total count of items (other than x) in S is less than $1/k$ fraction of the total count of all elements. If such a good set S exists, indeed x will be the majority element for that set S and will be identified.

In order to control false positives, we would like to ensure that items that are reported by this scheme have high frequency. Ideally, we would like to guarantee that all items reported have frequency at least $1/k$. We relax this slightly to say that for each item reported by this scheme, with high probability, it has frequency at least $(1 - \varepsilon)/k$. In order to ensure this, before reporting x as a hot item, we check that the count of each set containing x is at least $1/k$ fraction of the total count. Clearly, this condition will be satisfied if x is a hot item. Suppose x has frequency less than $(1 - \varepsilon)/k$. Consider a subset S containing x . We say that S is *light* for x if the total count of all items (other than x) in S is less than ε/k . Now if x has frequency less than $(1 - \varepsilon)/k$ and a set S is light for x , then x will not be reported as a hot item (because the count of elements in S is less than $1/k$ fraction of the total count). To control false positives, we would like to ensure that for any item x , there is at least one light set for x . (Note that requiring the existence of a light set is a stronger condition than requiring the existence of a good set.)

How do we construct such a collection of subsets? In fact, the subsets are constructed by a randomized algorithm which ensures the above properties with high probability. In other words, for any set of at most k hot items, the randomized construction of subsets ensures that with probability $1 - \delta$, every hot item x has a good subset S containing it. In addition, for any item x , we ensure that with high probability, there is a light set for x . This implies that for each item reported by this scheme, with high probability, it has frequency at least $(1 - \varepsilon)/k$.

Roughly speaking, the subsets are constructed at random by placing elements in them with probability $\varepsilon/2k$. This has the effect that the expected count of all elements in the set is a $\varepsilon/2k$ fraction of the total count. More precisely, the subsets

are constructed using the universal hash functions given by Carter and Wegman. We fix a prime $p \in [m, 2m]$, draw a uniformly and at random from $[1, p - 1]$ and b uniformly and at random from $[0, p - 1]$. Let $W = 2k/\varepsilon$ for $\varepsilon \leq 1$ (assume that ε is chosen so that W is an integer). Consider the hash function $h_{a,b}(x) = ((ax + b \bmod p) \bmod W)$. This hash function satisfies the property that

Proposition 1 *Over all choices of a and b , for $x \neq y$, $\Pr[h_{a,b}(x) = h_{a,b}(y)] \leq \varepsilon/2k$.*

We use this hash function family to define the subsets $S_{a,b,i} = \{x | h_{a,b}(x) = i\}$. Note that there are $2k/\varepsilon$ such sets for every choice of a and b , and every element belongs to one of the $2k/\varepsilon$ sets. For every pair of items, for randomly chosen a and b , the probability that both items are in the same set $S_{a,b,i}$ is at most $\varepsilon/2k$. In fact, we choose $\log(k/\delta)$ such pairs a, b and construct $2k/\varepsilon$ sets corresponding to each of them. This gives a total of $(2k/\varepsilon)\log(k/\delta)$ sets.

We show that this construction has the desired probabilities with high probability.

Lemma 3 *With probability at least $1 - \delta$, every hot item belongs to at least one good subset.*

Proof Consider a hot item x . For a random choice of a, b , suppose $x \in S_{a,b,i}$. The expected total count of other items that land into the same set as $S_{a,b,i}$ is at most an $\varepsilon/2k$ fraction of the total count of all items. Now, with probability at most $\varepsilon/2 \leq 1/2$, the total count of all such elements is greater than a $1/k$ fraction of the total count—this follows from Markov’s inequality. In other words, with probability at least $1/2$, the total count of all elements other than x that fall into $S_{a,b,i}$ is at most a $1/k$ fraction of the total count. Thus, the set $S_{a,b,i}$ is good for x with probability at least $1/2$ (over the random choice of a, b). Since we pick $\log(k/\delta)$ independent random a, b pairs, the probability that none of them yield a good set for x is at most $(1/2)^{\log(k/\delta)} = \delta/k$. We consider this a failure for the hot item x . Now, there are at most k hot items. By the union bound, the probability that there is a failure for any of the hot items is at most $k \times (\delta/k) = \delta$. Hence, with probability at least $1 - \delta$, every hot item belongs to at least one good subset. \square

Lemma 4 *For any item x , with probability $1 - \delta/k$, there is a light set for x .*

Proof Consider an item x . We will show that there is no light set for x with probability at most δ/k . The proof is very similar to the previous one. For a random choice of a, b , suppose $x \in S_{a,b,i}$. The expected total count of other items that land into the same set as $S_{a,b,i}$ is at most an $\varepsilon/2k$ fraction of the total count of all items. Now, with probability at most $1/2$, the total count of all such elements is greater than a ε/k fraction of the total count—from Markov’s inequality. Thus, the set $S_{a,b,i}$ is light for x with probability at least $1/2$ (over the random choice of a, b). Since we pick $\log(k/\delta)$ independent random a, b pairs, the probability that none of them is light for x is at most $(1/2)^{\log(k/\delta)} = \delta/k$. \square

We now put all the pieces together and describe the operation of the final algorithm. We pick $T = \log(k/\delta)$ random pairs a, b as described before. These implicitly define $(2k/\varepsilon) \log(k/\delta)$ subsets of items. For each such subset, we maintain $1 + \lceil \log m \rceil$ counters similar to the majority identification scheme discussed earlier. For each subset, we have one counter corresponding to each bit position and one corresponding to all the items in the subset. In addition, we have one common counter C for the total count of all the items $1, \dots, m$. The purpose of this common counter is to help reduce false positives.

When an item x is encountered, we determine the T sets it belongs to by evaluating the hash function $h_{a,b}(x)$ for the T pairs a, b . For each such set, we update the $1 + \lceil \log m \rceil$ counters for that set to reflect an insertion or a deletion of x as the case may be. In addition, we update the common counter C for all items.

As explained before, the subset construction ensures that with high probability, every hot item is contained in a good subset. If indeed a hot item is contained in a good subset x , it is the majority element in that subset and will be detected correctly by the majority detection scheme for that subset. The potential problem is that several items other than the hot items may also be reported. The subset construction ensures that for every item x , with high probability, there is a light set for x . Recall that before reporting a hot item, we check that the count for each set containing the item is at least a $1/k$ fraction of the total count. This ensures that for every item reported, with high probability, the frequency is at least $(1 - \varepsilon)/k$.

In order to further alleviate false positives, we can perform some simple tests to eliminate some of the sets which are not good. For every subset S , recall that we maintain a set of counters C_0, \dots, C_r . In addition, we have a common counter C . If S is a good set, we claim that for every $j \in [1, r]$, exactly one of C_j and $C_0 - C_j$ must be greater than C/k . We first explain why the failure of this condition implies that S is not a good set. If neither of C_j and $C_0 - C_j$ is greater than C/k then clearly S does not contain a hot item. Further, if both of them are greater than C/k , then either S contains two hot items, or S contains only one hot item but the sum of counts of the remaining items is greater than a $1/k$ fraction of the total count. In all these cases, S is not a good set.

5 Lower Bounds

In this section, we describe lower bounds on the space requirements for computing frequent elements exactly. These bounds were obtained by Alon, Matias and Szegedy [1] using a theorem from communication complexity. Informally, communication complexity studies a setting where two parties A with input x and B with input y want to jointly compute a function $f(x, y)$. Communication complexity measures the minimum number of bits that need to be exchanged by A and B in order to compute $f(x, y)$. This is a commonly used paradigm to show lower bounds on the space requirements of streaming algorithms. Imagine the input stream to be broken into two parts—the first half x and the second half y . Then the final output

of the streaming algorithm is a function of both x and y —call this $f(x, y)$. Consider two players A and B as follows: A runs the algorithm on x (the first half of the input) and B runs the algorithm on y (the second half of the input) using the output produced by A. The size of the storage space controls the communication from A to B (there is no communication in the other direction). Hence, a lower bound on the communication complexity of $f(x, y)$ is a lower bound on the storage space required by the algorithm.

We first give a simple counting argument to show that a deterministic algorithm that outputs the most frequent element (exactly) requires $\Omega(m)$ space. This illustrates the basic idea underlying the lower bound for randomized algorithms.

Theorem 3 *Any deterministic algorithm to produce the most frequent element exactly requires $\Omega(m)$ bits of storage.*

Proof Suppose we have a deterministic algorithm to determine the most frequent element using s bits. We will show that $s \geq m$. Consider the input stream to be constructed as follows: Let S be a set of $m/2$ elements from $[1, m]$, and let x_1, x_2 be a pair of elements from $[1, m]$ such that exactly one of x_1, x_2 is in S . The input consists of the elements of S in arbitrary order followed by the elements x_1, x_2 . The input is constructed so that one of x_1 or x_2 is the most frequent element in the data stream (depending on which of $x_1, x_2 \in S$) since this element occurs twice and all other elements occur at most once. We claim that any deterministic algorithm that correctly identifies the repeated element for all inputs of the form (S, x_1, x_2) must use $\Omega(m)$ bits. Consider the state of the storage after the $m/2$ elements of S have been seen. We claim that the state of the storage must be different for every possible set S . This would imply that $2^s \geq \binom{m}{m/2}$, and hence $s = \Omega(m)$.

Why should the state of the storage be different for every set S ? Suppose the state is the same for two distinct sets S_1, S_2 . Then consider $x_1 \in S_1 - S_2$ and $x_2 \in S_2 - S_1$. The input (S_1, x_1, x_2) has x_1 as the most frequent element, while the input (S_2, x_1, x_2) has x_2 as the most frequent element. Consider the operation of the algorithm on the two inputs (S_1, x_1, x_2) and (S_2, x_1, x_2) . By our assumption, after either S_1 or S_2 is processed initially, the state of the storage is the same. Beyond this point, the algorithm sees the same input (x_1, x_2) . Hence the final output of the algorithm is the same for both inputs. This means that the algorithm makes a mistake on one of the inputs. \square

The basic idea in the above proof was to construct the input from two sets: a set S of $m/2$ elements and a two element set $\{x_1, x_2\}$ which intersects S in exactly one element. Determining the most frequent element for this data stream amounts to computing the intersection of these two sets. A lower bound on the storage space can be derived by considering a communication complexity setting with two players, A and B, where player A has the first set, player B has the second set and the goal is to compute the intersection of the two sets. We showed that computing this intersection correctly for all such inputs requires $\Omega(m)$ bits of communication from A to B. This in turn means that a deterministic algorithm for frequent elements

needs $\Omega(m)$ bits of storage space. How about randomized algorithms for computing frequent elements? The same proof idea works—one can construct inputs by concatenating the elements of two sets such that estimating the most frequent element for those inputs amounts to computing the intersection of the two sets. A theorem of Kalyanasundaram and Schnitger [6] shows that computing such an intersection requires $\Omega(m)$ bits of communication.¹ Alon, Matias and Szegedy [1] used the result of [6] to show the following bound for randomized algorithms.

Theorem 4 *Any randomized algorithm to produce the most frequent element correctly with high probability requires $\Omega(m)$ bits of storage.*

We also describe an argument along similar lines given by Karp, Shenker and Papadimitriou [7], which shows that any streaming algorithm that computes hot items exactly, i.e., identifies exactly the set of items with counts exceeding n/k , must use $\Omega(m \log(n/m))$ space. In this proof, we assume that $n > 4m > 16k$.

Theorem 5 *Any streaming algorithm that identifies exactly the set of items with counts exceeding n/k , needs $\Omega(m \log(n/m))$ bits of storage in the worst case.*

Proof We will construct a large set K of sequences of length n containing m distinct items such that no item occupies more than $1/k$ fraction of the sequence. Consider the operation of the streaming algorithm on the sequences in K . We claim that the streaming algorithm must reach a distinct memory configuration for each of the sequences in K . Suppose to the contrary, there are two distinct sequences P and Q that result in the same configuration. Then there must be some element x that occurs a different number of times in P and Q . It is then possible to construct a sequence R that can be appended to P and Q , such that exactly one of the sequences PR and QR have x as a hot item, i.e., an item that occupies more than $1/k$ fraction of the sequence. Since the algorithm reached the same memory configuration on processing P and Q , it must produce the same output for the sequences PR and QR and hence is incorrect on some input.

This argument implies that the storage space used by the algorithm is $\Omega(\log |K|)$. In fact such a set K can be constructed such that $|K| \geq (\lfloor (n/2m) \rfloor)^{m-1}$. This implies a lower bound of $\Omega(m \log(n/m))$ on the storage space. \square

References

1. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.* **58**(1), 137–147 (1999)

¹Strictly speaking, their lower bound applies to the decision problem of checking whether the two sets intersect or not. However, it is easy to transform pairs of sets into an input for the frequent items algorithm such that the most frequent item is different depending on whether the sets intersect or not.

2. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams. *Theor. Comput. Sci.* **312**(1), 3–15 (2004)
3. G. Cormode, S. Muthukrishnan, What’s hot and what’s not: tracking most frequent items dynamically, in *Proceedings of PODS* (2003), pp. 296–306
4. G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, in *Proceedings of LATIN* (2004), pp. 29–38
5. E.D. Demaine, A. López-Ortiz, J.I. Munro, Frequency estimation of internet packet streams with limited space, in *Proceedings of ESA* (2002), pp. 348–360
6. B. Kalyanasundaram, G. Schnitger, The probabilistic communication complexity of set intersection. *SIAM J. Discrete Math.* **5**(4), 545–557 (1992)
7. R.M. Karp, S. Shenker, C.H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.* **28**, 51–55 (2003)
8. G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in *Proceedings of VLDB* (2002), pp. 346–357
9. J. Misra, D. Gries, Finding repeated elements. *Sci. Comput. Program.* **2**, 143–152 (1982)

Distinct-Values Estimation over Data Streams

Phillip B. Gibbons

1 Introduction

Estimating the number of distinct values in a data set is a well-studied problem with many applications [1–34]. The statistics literature refers to this as the problem of estimating the number of *species* or *classes* in a population (see [4] for a survey). The problem has been extensively studied in the database literature, for a variety of uses. For example, estimates of the number of distinct values for an attribute in a database table are used in query optimizers to select good query plans. In addition, histograms within the query optimizer often store the number of distinct values in each bucket, to improve their estimation accuracy [29, 30]. Distinct-values estimates are also useful for network resource monitoring, in order to estimate the number of distinct destination IP addresses, source-destination pairs, requested urls, etc. In network security monitoring, determining sources that send to many distinct destinations can help detect fast-spreading worms [12, 32].

Distinct-values estimation can also be used as a general tool for duplicate-insensitive counting: Each item to be counted views its unique id as its “value”, so that the number of distinct values equals the number of items to be counted. Duplicate-insensitive counting is useful in mobile computing to avoid double-counting nodes that are in motion [31]. It can also be used to compute the number of distinct neighborhoods at a given hop-count from a node [27] and the size of the transitive closure of a graph [7]. In a sensor network, duplicate-insensitive counting together with multi-path in-network aggregation enables robust and energy-efficient answers to count queries [8, 24]. Moreover, duplicate-insensitive counting is a building block for duplicate-insensitive computation of other aggregates, such as sum and average.

P.B. Gibbons (✉)

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: gibbons@cs.cmu.edu

stream S_1 : C, D, B, B, Z, B, B, R, T, S, X, R, D, U, E, B, R, T, Y, L, M, A, T, W
 stream S_2 : T, B, B, R, W, B, B, T, T, E, T, R, R, T, E, M, W, T, R, M, M, W, B, W

Fig. 1 Two example data streams of $N = 24$ items from a universe $\{A, B, \dots, Z\}$ of size $n = 26$. S_1 has 15 distinct values while S_2 has 6

More formally, consider a data set, S , of N items, where each item is from a universe of n possible values. Because multiple items may have the same value, S is a *multi-set*. The *number of distinct values* in S , called the *zeroth frequency moment* F_0 , is the number of values from the universe that occur at least once in S . In the context of this chapter, we will focus on the standard data stream scenario where the items in S arrive as an ordered sequence, i.e., as a data stream, and the goal is to estimate F_0 using only one pass through the sequence and limited working space memory. Figure 1 depicts two example streams, S_1 and S_2 , with 15 and 6 distinct values, respectively.

The data structure maintained in the working space by the estimation algorithm is called a *synopsis*. We seek an estimation algorithm that outputs an estimate \hat{F}_0 , a function of the synopsis, that is guaranteed to be close to the true F_0 for the stream. We focus on the following well-studied error metrics and approximation scheme:

- **relative error metric**—the *relative error* of an estimate \hat{F}_0 is $|\hat{F}_0 - F_0|/F_0$.
- **ratio error metric**—the *ratio error* of an estimate \hat{F}_0 is $\max(F_0/\hat{F}_0, \hat{F}_0/F_0)$.
- **standard error metric**—the *standard error* of an estimator Y with standard deviation $\sigma_Y(F_0)$ is $\sigma_Y(F_0)/F_0$.
- **(ϵ, δ) -approximation scheme**—an (ϵ, δ) -*approximation scheme* for F_0 is a randomized procedure that, given any positive $\epsilon < 1$ and $\delta < 1$, outputs an estimate \hat{F}_0 that is within a relative error of ϵ with probability at least $1 - \delta$.

Note that the standard error σ provides a means for an (ϵ, δ) trade-off: Under distributional assumptions (e.g., Gaussian approximation), the estimate is within $\epsilon = \sigma, 2\sigma, 3\sigma$ relative error with probability $1 - \delta = 65\%, 95\%, 99\%$, respectively. In contrast, the (ϵ, δ) -approximation schemes in this chapter do not rely on any distributional assumptions.

In this chapter, we survey the literature on distinct-values estimation. Section 2 discusses previous approaches based on sampling or using large space synopses. Section 3 presents the pioneering logarithmic space algorithm developed by Flajolet and Martin [13], as well as related extensions [1, 11]. We also discuss practical issues in using these algorithms in practice. Section 4 presents an algorithm that provides arbitrary precision ϵ [17], and a variant that improves on the space bound [2]. Section 5 gives lower bounds on the space needed to estimate F_0 for a data stream. Finally, Sect. 6 considers distinct-values estimation in a variety of important scenarios beyond the basic data stream set-up, including scenarios with selection predicates, deletions, sliding windows, distributed streams, and sensor networks. Table 1 summarizes the main algorithms presented in this chapter.

Table 1 Summary of the main algorithms presented in this chapter for distinct-values estimation over a data stream of values

Algorithm	Comment/Features
Linear counting [33]	linear space, very low standard error
FM [13], PCSA [13]	log space, good in practice, standard error
AMS [1]	realistic hash functions, constant ratio error
LogLog [11], Super-LogLog [11]	reduces PCSA synopsis space, standard error
Coordinated sampling [17]	(ϵ, δ) -approximation scheme
BJKST [2]	improved space bound, (ϵ, δ) -approximation scheme

2 Preliminary Approaches and Difficulties

In this section, we consider several previously studied approaches to distinct-values estimation and the difficulties with these approaches. We begin with previous algorithms based on random sampling.

2.1 Sampling-Based Algorithms

A common approach for distinct-values estimation from the statistics literature (as well as much of the early work in the database literature until the mid-1990s) is to collect a sample of the data and then apply sophisticated estimators on the distribution of the values in the sample [4, 5, 19–22, 25, 26]. This extended research focus on sampling-based estimators is due in part to three factors. First, in the applied statistics literature, the option of collecting data on more than a small sample of the population is generally not considered because of its prohibitive expense. For example, collecting data on every person in the world or on every animal of a certain species is not feasible. Second, in the database literature, where scanning an entire large data set is feasible, using samples to collect (approximate) statistics has proven to be a fast and effective approach for a variety of statistics [6]. Third, and most fundamental to the data streams context, the failings of existing sampling-based algorithms to accurately estimate F_0 (all known sampling-based estimators provide unsatisfactory results on some data sets of interest [5]) have spurred an ongoing focus on devising more accurate algorithms.

F_0 is a particularly difficult statistic to estimate from a sample. To gain intuition as to why this is the case, consider the following 33 % sample of a data stream of 24 items:

$$B, B, T, R, E, T, M, W.$$

Given this 33 % sample (with its 6 distinct values), does the entire stream have 6 distinct values, 18 distinct values (i.e., the 33 % sample has 33 % of the distinct values), or something in between? Note that this particular sample can be obtained by taking every third item of either S_1 (where $F_0 = 15$) or S_2 (where $F_0 = 6$) from

```

for  $i := 0, \dots, s - 1$  do  $M[i] := 0$ 
foreach (stream item with value  $v$ ) do
     $M[h(v)] := 1$ 
let  $z := |\{i : M[i] = 0\}|$ 
return  $s \ln \frac{s}{z}$ 

```

Fig. 2 The linear counting algorithm [33]

Fig. 1. Thus, despite sampling a large (33 %) percentage of the data, estimating F_0 remains challenging, because the sample can be viewed as fairly representative of either S_1 or S_2 —two streams with very different F_0 's. In fact, in the worse case, all sampling-based F_0 estimators are provably inaccurate: Charikar et al. [5] proved that estimating F_0 to within a small constant factor (with probability $> \frac{1}{2}$) requires (in the worst case) that *nearly the entire data set be sampled* (see Sect. 5).

Thus, any approach based on a uniform sample of say 1 % of the data (or otherwise reading just 1 % of the data) is unable to provide good guaranteed error estimates in either the worst case or in practice. Highly-accurate answers are possible only if (nearly) the entire data set is read. This motivates the need for an effective streaming algorithm.

2.2 Streaming Approaches

Clearly, F_0 can be computed *exactly* in one pass through the entire data set, by keeping track of *all* the unique values observed in the stream. If the universe is $[1..n]$, a bit vector of size n can be used as the synopsis, initialized to 0, where bit i is set to 1 upon observing an item with value i . However, in many cases the universe is quite large (e.g., the universe of IP addresses is $n = 2^{32}$), making the synopsis much larger than the stream size N ! Alternatively, we can maintain a set of all the unique values seen, which takes $F_0 \log_2 n$ bits because each value is $\log_2 n$ bits. However, F_0 can be as large as N , so again the synopsis is quite large. A common approach for reducing the synopsis by (roughly) a factor of $\log_2 n$ is to use a hash function $h()$ mapping values into a $\Theta(n)$ size range, and then adapting the aforementioned bit vector approach. Namely, bit $h(i)$ is set upon observing an item with value i . Whang et al. [33], for example, proposed an algorithm, called *linear counting*, depicted in Fig. 2. The hash function $h()$ in the figure maps each value v uniformly at random to a number in $[0, s - 1]$. They show that using a load factor $F_0/s = 12$ provides estimates to within a 1 % standard error. There are two limitations of this algorithm. First, we need to have a good *a priori* knowledge of F_0 in order to set the hash table size s . Second, the space is proportionate to F_0 , which can be as large as n . Note that the approximation error arises from collisions in the hash table, i.e., distinct values in the stream mapping to the *same* bit position. To help alleviate this source of error, Bloom filters (with their multiple hash functions) have been used effectively [23], although the space remains $\Theta(n)$ in the worst case.

```

for  $i := 0, \dots, L - 1$  do  $M[i] := 0$ 
foreach (stream item with value  $v$ ) do {
     $b :=$  the largest  $i \geq 0$  such that the  $i$  rightmost bits in  $h(v)$  are all 0
     $M[b] := 1$ 
}
let  $Z := \min\{i : M[i] = 0\}$  // i.e., the least significant 0-bit in  $M$ 
return  $\lfloor \frac{2^Z}{0.77351} \rfloor$ 

```

Fig. 3 The FM algorithm [13], using only a single hash function and bit vector

The challenge in estimating F_0 using $o(n)$ space is that because there is insufficient space to keep track of *all* the unique values seen, it is impossible to determine whether or not an arriving stream value increases the number of distinct values seen thus far.

3 Flajolet and Martin's Algorithm

In this section, we present Flajolet and Martin's (FM) pioneering algorithm for distinct-values estimation. We also describe a related algorithm by Alon, Matias, and Szegedy. Finally, we discuss practical issues and optimizations for these algorithms.

3.1 The Basic FM Algorithm

Over two decades ago, Flajolet and Martin [13] presented the first small space distinct-values estimation algorithm. Their algorithm, which we will refer to as the *FM algorithm*, is depicted in Fig. 3. In this algorithm, the $L = \Theta(\log(\min(N, n)))$ space synopsis consists of a bit vector M initialized to all 0, where N is the number of items and n is the size of the universe. The main idea of the algorithm is to let each item in the data set select at random a bit in M and set it to 1, with (quasi-)geometric distribution, i.e., $M[i]$ is selected with probability (\approx) $2^{-(i+1)}$. This selection is done using a hash function h that maps each value v uniformly at random to an integer in $[0, 2^L - 1]$, and then determining the largest b such that the b rightmost bits in $h(v)$ are all 0. In this way, each distinct value gets mapped to $b = i$ with probability $2^{-(i+1)}$. For the vector length L , it suffices to take any number $> \log_2(F_0) + 4$ [13]. As F_0 is unknown, we can conservatively use $L = \log_2(\min(N, n)) + 5$.

The intuition behind the FM algorithm is as follows. Using a hash function ensures that all items with the same value will make the same selection; thus the final bit vector M is independent of any duplications among the item values. For each distinct value, the bit b is selected with probability $2^{-(b+1)}$. Accordingly, we expect

Item value v	Hash Function 1		Hash Function 2		Hash Function 3	
	b	$M[\cdot]$	b	$M[\cdot]$	b	$M[\cdot]$
15	1	00000010	1	00000010	0	00000001
36	0	00000011	1	00000010	0	00000001
4	0	00000011	0	00000011	0	00000001
29	0	00000011	2	00000111	1	00000011
9	3	00001011	0	00000111	0	00000011
36	0	00001011	1	00000111	0	00000011
14	1	00001011	0	00000111	1	00000011
4	0	00001011	0	00000111	0	00000011

 $Z = 2$ $Z = 3$ $Z = 2$

$$\text{Estimate } \hat{F}_0 = \lfloor \frac{2^{(2+3+2)/3}}{0.77351} \rfloor = 6$$

Fig. 4 Example run of the FM algorithm on a stream of 8 items, using three hash functions. Each $M[\cdot]$ is depicted as an $L = 8$ bit binary number, with $M[0]$ being the rightmost bit shown. The estimate, 6, matches the number of distinct values in the stream

$M[b]$ to be set if there are at least 2^{b+1} distinct values. Because bit $Z - 1$ is set but not bit Z , there are likely greater than 2^Z but fewer than 2^{Z+1} distinct values. Flajolet and Martin's analysis shows that $E[Z] \approx \log_2(0.77351 \cdot F_0)$, so that $2^Z/0.77351$ is a good choice in that range.

To reduce the variance in the estimator, Flajolet and Martin take the average over tens of applications of this procedure (with different hash functions). Specifically, they take the average, \bar{Z} , of the Z 's for different hash functions and then compute $\lfloor 2^{\bar{Z}}/0.77351 \rfloor$. An example is given in Fig. 4.

The error guarantee, space bound, and time bound are summarized in the following theorem.

Theorem 1 ([13]) *The FM algorithm with k (idealized) hash functions produces an estimator with standard error $O(1/\sqrt{k})$, using $k \cdot L$ memory bits for the bit vectors, for any $L > \log_2(\min(N, n)) + 4$. For each item, the algorithm performs $O(k)$ operations on L -bit memory words.*

The space bound does not include the space for representing the hash functions. The time bound assumes that computing h and b are constant time operations. Sections 3.3 and 3.4 will present optimizations that significantly reduce both the space for the bit vectors and the time per item, without increasing the standard error.

3.2 The AMS Algorithm

Flajolet and Martin [13] analyze the error guarantees of their algorithm assuming the use of an explicit family of hash functions with ideal random properties (namely, that h maps each value v uniformly at random to an integer in the given range). Alon, Matias, and Szegedy [1] adapted the FM algorithm to use (more realistic) linear

Consider a universe $U = \{1, 2, \dots, n\}$. Let d be the smallest integer so that $2^d > n$

Consider the members of U as elements of the finite field $F = GF(2^d)$,

which are represented by binary vectors of length d

Let a and b be two random members of F , chosen uniformly and independently

Define $h(v) := a \cdot v + b$, where the product and addition are computed in the field F

$R := 0$

foreach (stream item with value v) do {

$b :=$ the largest $i \geq 0$ such that the i rightmost bits in $h(v)$ are all 0

$R := \max(R, b)$

}

return 2^R

Fig. 5 The AMS algorithm [1], using only a single hash function

hash functions. Their algorithm, which we will call the *AMS algorithm*, produces an estimate with provable guarantees on the ratio error. We discuss the AMS algorithm in this section.

First, note that in general, the final bit vector M returned by the FM algorithm in Sect. 3.1 consists of three parts, where the first part is all 0's, the third part is all 1's, and the second part (called the *fringe* in [13]) is a mix of 0's and 1's that starts with the most significant 1 and ends with the least significant 0. Let R (Z) be the position of the most (least) significant bit that is set to 1 (0, respectively). For example, for $M[\cdot] = 000101111$, $R = 5$ and $Z = 4$. Whereas the FM algorithm uses Z in its estimate, the AMS algorithm uses R . Namely, the estimate is 2^R . Figure 5 presents the AMS algorithm. Note that unlike the FM algorithm, the AMS algorithm does not maintain a bit vector but instead directly keeps track of the most significant bit position set to 1.

The space bound and error guarantees for the AMS algorithm are summarized in the following theorem.

Theorem 2 ([1]) *For every $r > 2$, the ratio error of the estimate returned by the AMS algorithm is at most r with probability at least $1 - 2/r$. The algorithm uses $\Theta(\log n)$ memory bits.*

Proof Let $b(v)$ be the value of b computed for $h(v)$. By the construction of $h()$, we have that for every fixed v , $h(v)$ is uniformly distributed over F . Thus the probability that $b(v) \geq i$ is precisely $1/2^i$. Moreover, for every fixed distinct v_1 and v_2 , the probability that $b(v_1) \geq i$ and $b(v_2) \geq i$ is precisely $1/2^{2i}$.

Fix an i . For each element $v \in U$ that appears at least once in the stream, let $W_{v,i}$ be the indicator random variable whose value is 1 if and only if $b(v) \geq i$. Let $Z_i = \sum W_{v,i}$, where v ranges over all the F_0 elements v that appear in the stream. By linearity of expectation and because the expectation of each $W_{v,i}$ is $1/2^i$, the expectation $E[Z_i]$ of Z_i is $F_0/2^i$. By pairwise independence, the variance of Z_i is $F_0 \frac{1}{2^i} (1 - \frac{1}{2^i}) < F_0/2^i$. Therefore, by Markov's Inequality, if $2^i > rF_0$ then $\Pr[Z_i > 0] < 1/r$, since $E[Z_i] = F_0/2^i < 1/r$. Similarly, by Chebyshev's Inequality, if $r2^i < F_0$ then $\Pr[Z_i = 0] < 1/r$, since $\text{Var}[Z_i] < F_0/2^i = E[Z_i]$ and

hence $\Pr[Z_i = 0] \leq \text{Var}[Z_i]/(\mathbb{E}[Z_i]^2) < 1/\mathbb{E}[Z_i] = 2^i/F_0$. Because the algorithm outputs $\hat{F}_0 = 2^R$, where R is the maximum i for which $Z_i > 0$, the two inequalities above show that the probability that the ratio between \hat{F}_0 and F_0 is not between $1/r$ and r is smaller than $2/r$, as needed.

As for the space bound, note that the algorithm uses $d = O(\log n)$ bits representing an irreducible polynomial needed in order to perform operations in F , $O(\log n)$ bits representing a and b , and $O(\log \log n)$ bits representing the current maximum R value. \square

The probability of a given ratio error can be reduced by using multiple hash functions, at the cost of a linear increase in space and time.

Corollary 1 *With k hash functions, the AMS algorithm uses $O(k \log n)$ memory bits and, for each item, performs $O(k)$ operations on $O(\log n)$ -bit memory words.*

The space bound includes the space for representing the hash functions. The time bound assumes that computing $h(v)$ and $b(v)$ are constant time operations.

3.3 Practical Issues

Although the AMS algorithm has stronger error guarantees than the FM algorithm, the FM algorithm is somewhat more accurate in practice, for a given synopsis size [13]. In this section we argue why this is the case, and discuss other important practical issues.

First, the most significant 1-bit (as is used in the AMS algorithm) can be set by a single outlier hashed value. For example, suppose a hashed value sets the i th bit whereas all other hashed values set bits $\leq i - 3$, so that $M[\cdot]$ is of the form 0001001111. In such cases, it is clear from M that 2^i is a significant overestimate of F_0 : it is not supported by the other bits (in particular, both bits $i - 1$ and $i - 2$ are 0's not 1's). On the other hand, the least significant 0-bit (as is used in the FM algorithm) is supported by all the bits to its right, which must be all 1's. Thus FM is more robust against single outliers.

Second, although the AMS algorithm requires only $\approx \log \log n$ bits to represent the current R , while the FM algorithm needs $\approx \log n$ bits to represent the current $M[\cdot]$, this space savings is insignificant compared to the $O(\log n)$ bits the AMS algorithm uses for the hash function computations. In practice, the same class of hash functions (i.e., $h(v) = a \cdot v + b$) can be used in the FM algorithm, so that it too uses $O(\log n)$ bits per hash function.

Note that there are common scenarios where the space for the hash functions is not as important as the space for the accumulated synopsis (i.e., R or $M[\cdot]$). For example, in the distributed streams scenario (discussed in Sect. 6), an accumulated synopsis is computed for each stream locally. In order to estimate the total number of distinct values across all streams, the current accumulated synopses are collected.

$M_j[\cdot]$'s:	00001111, 00001011, 00000111, 00010111, 00011111, 00001111
Interleaved:	00000000000000000000110110011101111111111111111111111111111111
End-encoded:	11011001110, 16

Fig. 6 Example FM synopsis compression, for $k = 6$ and $L = 8$

Thus the message size depends only on the accumulated synopsis size. Similarly, when distinct-values estimation techniques are used for duplicate-insensitive aggregation in sensor networks (see Sect. 6), the energy used in sending messages depends on the accumulated synopsis size and not the hash function size. In such scenarios, the $(\log \log n)$ -bit AMS synopses are preferable to the $(\log n)$ -bit FM synopses.

Note, however, that the size of the FM accumulated synopsis can be significantly reduced, using the following simple compression technique [8, 13, 27]. Recall that at any point during the processing of a stream, $M[\cdot]$ consists of three parts, where the first part is all 0's, the second part (the *fringe*) is a mix of 0's and 1's, and the third part is all 1's. Recall as well that multiple $M[\cdot]$ are constructed for a given stream, each using a distinct hash function. These $M[\cdot]$ are likely to share many bits in common, because each $M[\cdot]$ is constructed using the same algorithm on the same data. Suppose we are using k hash functions, and for $j = 1, \dots, k$, let $M_j[\cdot]$ be the bit vector created using the j th hash function. The naive approach of concatenating $M_1[\cdot]$, $M_2[\cdot]$, \dots , $M_k[\cdot]$ uses $k \cdot L$ bits (recall that each $M_j[\cdot]$ is L bits), which is $\approx k \log_2 n$ bits. Instead, we will interleave the bits, as follows:

$$\begin{aligned} M_1[L-1], M_2[L-1], \dots, M_k[L-1], M_1[L-2], \dots, \\ M_k[L-2], \dots, M_1[0], \dots, M_k[0], \end{aligned}$$

and then run-length encode the resulting bit vector's prefix and suffix. From the above arguments, the prefix of the interleaved bit vector is all 0's and the suffix is all 1's. We use a simple encoding that ignores the all 0's prefix, consisting of (i) the interleaved bits between the all 0's prefix and the all 1's suffix and (ii) a count of the length of the all 1's suffix. An example is given in Fig. 6. Note that the interleaved fringe starts with the maximum bit set among the k fringes and ends with their minimum unset bit. Each fringe is likely to be tightly centered around the current $\log_2 F_0$, e.g., for a given hash function and $c > 0$, the probability that no bit larger than $\log_2 F_0 + c$ is set is $(1 - 2^{-(\log_2 F_0 + c)})^{F_0} \approx e^{-1/2^c}$. At any point in the processing of the stream, a similar argument shows that the interleaved fringe is expected to be fewer than $O(k \log k)$ bits. Thus our simple encoding is expected to use fewer than $\log_2 \log_2 n + O(k \log k)$ bits. In comparison, the AMS algorithm uses $k \log_2 \log_2 n$ bits for its accumulated synopsis, although this too can be reduced through compression techniques.

The number of bit vectors, k , in the FM algorithm is a tunable parameter trading off space for accuracy. The relative error as a function of k has been studied empirically in [8, 13, 24, 27] and elsewhere, with $< 15\%$ relative error reported for $k = 20$ and $< 10\%$ relative error reported for $k = 64$, on a variety of data sets. These studies

```

for  $j := 1, \dots, k$  and  $i := 0, \dots, L - 1$  do  $M_j[i] := 0$ 
foreach (stream item with value  $v$ ) do {
     $x := h(v) \bmod k$  // Note:  $k$  is a power of 2
     $b :=$  the largest  $i \geq 0$  such that the  $i$  rightmost bits in  $[h(v)/k]$  are all 0
     $M_x[b] := 1$ 
}
 $\bar{Z} := \frac{1}{k} \sum_{j=1}^k \min\{i : M_j[i] = 0\}$ 
return  $\lfloor \frac{k}{0.77351} 2^{\bar{Z}} \rfloor$ 

```

Fig. 7 The PCSA algorithm [13]

show a strong diminishing return for increases in k . Theorem 1 shows that the standard error is $O(1/\sqrt{k})$. Thus reducing the standard error from 10 % to 1 % requires increasing k by a factor of 100! In general, to obtain a standard error at most ϵ we need $k = \Theta(1/\epsilon^2)$. Estan, Varghese and Fisk [12] present a number of techniques for further improving the constants in the space vs. error trade-off, including using multi-resolution and adaptive bit vectors.

Both the FM and AMS algorithms use the largest i such that the i rightmost bits in $h(v)$ are all 0, in order to create an exponential distribution onto an integer range. A related, alternative approach by Cohen [7] is to (i) use a hash function that maps uniformly to the interval $[0, 1]$, (ii) maintain the minimum hashed value x seen thus far in the stream S (i.e., $x = \min_{v \in S} \{h(v)\}$), and then (iii) return $\frac{1}{x} - 1$ as the estimate for F_0 . The intuition is that if there are F_0 distinct values mapped uniformly at random to $[0, 1]$, then we may expect them to divide the interval into $F_0 + 1$ relatively evenly-spaced subintervals, i.e., subintervals of size $\frac{1}{F_0+1}$. As $[0, x]$ is the first such subinterval, $x = \frac{1}{F_0+1}$, and hence $\frac{1}{x} - 1$ is used as the estimate for F_0 . As with the FM and AMS algorithms, the error guarantee can be improved by taking multiple hash functions and averaging. Empirically, this approach is not as accurate as the FM algorithm for a given synopsis size [27].

3.4 Improving the Per-Item Processing Time

The FM algorithm as presented in Sect. 3.1 performs $O(k)$ operations on memory words of $\approx \log_2 n$ bits (recall Theorem 1) for each stream item. This is because a different hash function is used for each of the k bit vectors. To reduce the processing time per item from $O(k)$ to $O(1)$, Flajolet and Martin present the following variant on their algorithm, called *Probabilistic Counting with Stochastic Averaging (PCSA)*.

In the PCSA algorithm (see Fig. 7), k bit vectors are used (for k a power of 2) but only a single hash function $h()$. For each stream item with value v , the $\log_2 k$ least significant bits of $h(v)$ are used to select a bit vector. Then the remaining $L - \log_2 k$ bits of $h(v)$ are used to select a position within that bit vector, according to an exponential distribution (as in the basic FM algorithm). To compute an estimate, PCSA averages over the positions of the least significant 0-bits, computes 2 to the

```

for  $j := 1, \dots, k$  do  $R_j := 0$ 
foreach (stream item with value  $v$ ) do {
     $x := h(v) \bmod k$  // Note:  $k$  is a power of 2
     $b :=$  the largest  $i \geq 0$  such that the  $i$  rightmost bits in  $[h(v)/k]$  are all 0
     $R_x := \max(R_x, b)$ 
}
 $\bar{Z} := \frac{1}{k} \sum_{j=1}^k R_j$ 
return  $\lfloor (0.79402k - 0.84249)2^{\bar{Z}} \rfloor$ 

```

Fig. 8 The LogLog algorithm [11]

power of that average, and divides by the bias factor 0.77351. To compensate for the fact that each bit vector has seen only $1/k$ 'th of the distinct items on average, the estimate is multiplied by k .

The error guarantee, space bound, and time bound are summarized in the following theorem.

Theorem 3 ([13]) *The PCSA algorithm with k bit vectors and an (idealized) hash function produces an estimator with standard error $0.78/\sqrt{k}$, using $k \cdot L$ memory bits for the bit vectors, for any $L > \log_2(\min(N, n)/k) + 4$. For each item, the algorithm performs $O(1)$ operations on $(\log_2 n)$ -bit memory words.*

The space bound does not include the space for representing the hash function. The time bound assumes that computing h and b are constant time operations. Although some of the operations use only L -bit words, the word size is dominated by the $\log_2 n$ bits for the item value v .

Durand and Flajolet [11] recently presented a variant of PCSA, called the *LogLog* algorithm, that reduces the size of the accumulating synopsis from $\log n$ to $\log \log n$. (As in FM and PCSA, the size of the hash function is not accounted for in the space bound.) The algorithm, depicted in Fig. 8, differs from PCSA by maintaining the maximum bit set (as in AMS) and using a different function for computing the estimate. The analysis in [11] gives the bias correction factor $(0.79402k - 0.84249)$, where k is the number of maximums (i.e., R_j 's) maintained. With k maximums, the standard error is shown to be $1.30/\sqrt{k}$ (assuming idealized hash functions). This is higher than with the PCSA algorithm (supporting our earlier argument that the least-significant 0-bit is more accurate than the most-significant 1-bit), but the synopsis size is smaller (when ignoring the hash function space and not using the compression tricks in Sect. 3.3).

Durand and Flajolet [11] also present a further improvement, called *SuperLogLog*, that differs from LogLog in two aspects. First, it discards the largest 30 % of the estimates, in order to decrease the variance. As discussed in Sect. 3.3, using the maximum bit set is subject to overestimation caused by outlier bits being set. By discarding the largest estimates, these outliers are discarded. Note that a different correction factor is needed in order to compensate for this additional source of bias [11]. Second, it represents each maximum R_j using only

$L = \log_2(\log_2(n/k) + 3)$ bits, and again corrects for the additional bias. The error guarantee, space bound, and time bound are summarized in the following theorem.

Theorem 4 ([11]) *The Super-LogLog algorithm with k maximums and an (idealized) hash function produces an estimator with standard error $1.05/\sqrt{k}$, using $k \cdot L$ memory bits for the maximums, where $L = \lceil \log_2 \lceil \log_2(\min(N, n)/k) + 3 \rceil \rceil$. For each item, the algorithm performs $O(1)$ operations on $(\log_2 n)$ -bit memory words.*

The space bound does not include the space for representing the hash function. The time bound assumes that computing h and b are constant time operations. Although some of the operations use only L -bit words, the word size is dominated by the $\log_2 n$ bits for the stream value v . The standard error $1.05/\sqrt{k}$ for Super-LogLog is higher than the $0.78/\sqrt{k}$ error for PCSA. Comparing the synopsis sizes (ignoring the hash functions), super-LogLog uses a fixed $\approx k \log_2 \log_2(n/k)$ bits, whereas PCSA using the compression tricks of Sect. 3.3 uses an expected $\approx \log_2 \log_2 n + O(k \log k)$ bits.

4 (ϵ, δ) -Approximation Schemes

None of the algorithms presented thus far provides the strong guarantees of an (ϵ, δ) -approximation scheme. In this section, we present two such algorithms: the Coordinated Sampling algorithm of Gibbons and Tirthapura [17], and an improvement by Bar-Yossef et al. [2] that achieves near optimal space.

4.1 Coordinated Sampling

Gibbons and Tirthapura [17] gave the first (ϵ, δ) -approximation scheme for F_0 . Their algorithm, called *Coordinated Sampling*, is depicted in Fig. 9.

In the algorithm, there are $k = \Theta(\log(1/\delta))$ instances of the same procedure, differing only in their use of different hash functions $h_j()$. For each instance, the hash function is used to assign each potential stream value to a “level”, such that half the values are assigned to level 0, a quarter to level 1, etc. The algorithm maintains a set of the $\approx \tau$ distinct stream values that have the highest levels among those observed thus far. More specifically, it keeps track of the minimum level ℓ_j such that there are at most τ distinct stream values with level at least ℓ_j , as well as the set, S_j , of these stream values.

As in the AMS algorithm (Sect. 3.2), any uniform pairwise independent hash function can be used for $h_j()$; for example, linear hash functions can be used. Let $b_j(v)$ be the value of b computed in the algorithm for $h_j(v)$. Following the argument in Sect. 3.2, we have that $\Pr\{b_j(v) = \ell\} = \frac{1}{2^{\ell+1}}$ and $\Pr\{b_j(v) \geq \ell\} = \frac{1}{2^\ell}$ for $\ell = 0, \dots, \log n - 1$. Hence, S_j is always a uniform random sample of the distinct stream

```

for  $j := 1, \dots, k$  do {  $\ell_j := 0, S_j := \emptyset$  }
foreach (stream item with value  $v$ ) do {
    for  $j := 1, \dots, k$  do {
         $b :=$  the largest  $i \geq 0$  such that the  $i$  rightmost bits in  $h_j(v)$  are all 0
        if  $b \geq \ell_j$  and  $(v, b) \notin S_j$  do {
             $S_j := S_j \cup \{(v, b)\}$ 
            // if  $S_j$  is too large, discard the level  $\ell_j$  sample points from  $S_j$ 
            while  $|S_j| > \tau$  do {
                 $S_j := S_j - \{(v', b') : b' = \ell_j\}$ 
                 $\ell_j := \ell_j + 1$ 
            }
        }
    }
}
return median $j=1, \dots, k$ ( $|S_j| \cdot 2^{\ell_j}$ )

```

Fig. 9 The Coordinated Sampling algorithm [17]. The values of k and τ depend on the desired ϵ and δ : $k = 36 \log_2(1/\delta)$ and $\tau = 36/\epsilon^2$, where the constant 36 is determined by the worst case analysis and can be much smaller in practice

values observed thus far, where each value is in the sample with probability $2^{-\ell_j}$. Thus, Coordinated Sampling uses $|S_j| \cdot 2^{\ell_j}$ as the estimate for the number of distinct values in the stream. To ensure that the estimate is within ϵ with probability $1 - \delta$, it computes the median over $\Theta(\log(1/\delta))$ such estimates.

Each step of the algorithm within the “for” loop can be done in constant (expected) time by maintaining the appropriate data structures, assuming (as we have for the previous algorithms) that computing hash functions and determining b are constant time operations. For example, each S_j can be stored in a hash table T_j of 2τ entries, where the pair (v, b) is the hash key. This enables both tests for whether a given (v, b) is in S_j and insertions of a new (v, b) into S_j to be done in constant expected time. We can enable constant time tracking of the size of S_j by maintaining an array of $\log n + 1$ “level” counters, one per possible level, which keep track of the number of pairs in S_j for each level. We also maintain a running count of the size of S_j . This counter is incremented by 1 upon insertion into S_j and decremented by the corresponding level counter upon deleting all pairs in a level. In the latter case, in order to quickly delete from S_j all such pairs, we leave these deleted pairs in place, removing them lazily as they are encountered in subsequent visits to T_j . (We need not explicitly mark them as deleted because subsequent visits see that their level numbers are too small and treat them as deleted.)

The error guarantee, space bound, and time bound are summarized in the following theorem. The space bound includes the space for representing the hash functions. The time bound assumes that computing hash functions and b are constant expected time operations.

Theorem 5 ([17]) *The Coordinated Sampling algorithm provides an (ϵ, δ) -approximation scheme, using $O(\frac{\log n \log(1/\delta)}{\epsilon^2})$ memory bits. For each item, the algorithm performs an expected $O(\log(1/\delta))$ operations on $(\log_2 n)$ -bit memory words.*

Proof We have argued above about the time bound. The space bound is $O(k \cdot \tau)$ memory words, i.e., $O(\frac{\log n \log(1/\delta)}{\epsilon^2})$ memory bits.

In what follows, we sketch the proof that Coordinated Sampling is indeed an (ϵ, δ) -approximation scheme. A difficulty in the proof is that the algorithm decides when to stop changing levels based on the outcome of random trials, and hence may stop at an incorrect level, and make correspondingly bad estimates. We will argue that the probability of stopping at a “bad” level is small, and can be accounted for in the desired error bound.

Accordingly, consider the j th instance of the algorithm. For $\ell \in \{0, \dots, \log n\}$ and $v \in \{1, \dots, n\}$, we define the random variables $X_{\ell, v}$ such that $X_{\ell, v} = 1$ if v 's level is at least ℓ and 0 otherwise. For the stream S , we define $X_\ell = \sum_{v \in S} X_{\ell, v}$ for every level ℓ . Note that after processing S , the value of ℓ_j is the lowest numbered level f such that $X_f \leq \tau$. The algorithm uses the estimate $2^f \cdot X_f$.

For every level $\ell \in \{0, \dots, \log n\}$, we define B_ℓ such that $B_\ell = 1$ if $2^\ell X_\ell \notin [(1 - \epsilon)F_0, (1 + \epsilon)F_0]$ and 0 otherwise. Level ℓ is “bad” if $B_\ell = 1$, and “good” otherwise. Let E_ℓ denote the event that the final value of ℓ_j is ℓ , i.e., that f equals ℓ . The heart of the proof is to show the following:

$$\Pr\{\text{Given instance produces an estimate not in } [(1 - \epsilon)F_0, (1 + \epsilon)F_0]\} < \frac{1}{3}. \quad (1)$$

Let P be the probability in Eq. (1). Let ℓ^* denote the first level such that $E[X_{\ell^*}] \leq \frac{2}{3}\tau$. The instance produces an estimate not within the target range if B_f is true for the level f such that E_f is true. Thus,

$$P = \sum_{i=0}^{\log n} \Pr\{E_i \wedge B_i\} < \sum_{i=0}^{\ell^*} \Pr\{B_i\} + \sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\}. \quad (2)$$

The idea behind using the inequality to separate the B_i terms from the E_i terms is that the lower levels (until ℓ^*) are likely to have good estimates and the algorithm is unlikely to keep going beyond level ℓ^* .

As in the proof for the AMS algorithm (Theorem 2), we have that for $\ell = 0, \dots, \log n - 1$,

$$\mathbb{E}[X_\ell] = \frac{F_0}{2^\ell}, \quad (3)$$

and

$$\text{var}[X_\ell] < \frac{F_0}{2^\ell}. \quad (4)$$

We will now show that

$$\sum_{i=0}^{\ell^*} \Pr\{B_i\} < \frac{6}{\epsilon^2 \tau}. \quad (5)$$

To see this, we first express $\Pr\{B_i\}$ in terms of Eq. (3): $\Pr\{B_i\} = \Pr\{|X_i - \frac{F_0}{2^i}| \geq \epsilon \frac{F_0}{2^i}\}$. Then, from Eq. (4) and using Chebyshev's inequality, we have $\Pr\{B_i\} < \frac{2^i}{F_0 \epsilon^2}$.

Hence, $\sum_{i=0}^{\ell^*} \Pr\{B_i\} < \sum_{i=0}^{\ell^*} \frac{2^i}{F_0 \epsilon^2} = \frac{2^{\ell^*+1}}{F_0 \cdot \epsilon^2}$. Now, because ℓ^* is the first level such that $\frac{F_0}{2^{\ell^*}} \leq \frac{2}{3}\tau$, we have that $F_0 > 2^{\ell^*-1} \cdot \frac{2}{3}\tau$. Thus, $\sum_{i=0}^{\ell^*} \Pr\{B_i\} < \frac{2^{\ell^*+1}}{F_0 \cdot \epsilon^2} < \frac{6}{\epsilon^2 \tau}$, establishing Eq. (5).

Next, we will show that

$$\sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} < \frac{6}{\tau}. \quad (6)$$

To see this, we first observe that $\sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} = \Pr\{X_{\ell^*} > \tau\}$, because the E_i 's are mutually exclusive. Because $E[X_{\ell^*}] < \frac{2}{3}\tau$, we have $\Pr\{X_{\ell^*} > \tau\} < \Pr\{X_{\ell^*} - E[X_{\ell^*}] > \frac{\tau}{3}\}$. By Chebyshev's inequality and Eq. (4), this latter probability is less than $\frac{9}{\tau^2} \cdot \frac{F_0}{2^{\ell^*}}$. Plugging in the fact that $\frac{2}{3}\tau > E[X_{\ell^*}] = \frac{F_0}{2^{\ell^*}}$, we obtain $\sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} < \frac{9}{\tau^2} \cdot \frac{2\tau}{3}$, establishing Eq. (6).

Plugging into Eq. (2) the results from Eqs. (5) and (6), and setting $\tau = 36/\epsilon^2$, we have $P < \frac{1}{6} + \frac{\epsilon^2}{6} < \frac{1}{3}$. Thus, Eq. (1) is established.

Finally, the median fails to be an (ϵ, δ) -estimator of F_0 if at least $k/2$ instances of the algorithm fail. By Eq. (1), we expect $< k/3$ to fail, and hence by Chernoff bounds, the probability the algorithm fails is less than $\exp(-k/36)$. Setting $k = 36 \log(1/\delta)$ makes this probability less than δ , completing the proof of the theorem. \square

4.2 Improving the Space Bound

Bar-Yossef et al. [2] showed how to adapt the Coordinated Sampling algorithm in order to improve the space bound. Specifically, their algorithm, which we call the *BJKST algorithm*, stores the elements in S_j using less space, as follows. Instead of storing the pair (v, b) , as in Coordinated Sampling, the BJKST algorithm stores $g(v)$, for a suitably chosen hash function $g()$. Namely, $g()$ is a (randomly chosen) uniform pairwise independent hash function that maps values from $[0..n - 1]$ to the range $[0..R - 1]$, where $R = 3((\log n + 1)\tau)^2$. Thus only $O(\log \log n + \log(1/\epsilon))$ bits are needed to store $g(v)$. The level b for v is represented implicitly by storing the hashed values as a collection of balanced binary search trees, one tree for each level.

The key observation is that for any given instance of the algorithm, $g()$ is applied to at most $(\log n + 1) \cdot \tau$ distinct values. Thus, the choice of R ensures that with probability at least $5/6$, $g()$ is injective on these values. If $g()$ is indeed injective, then using $g()$ did not alter the basic progression of the instance. The alternative occurs with probability at most $1/6$. To compensate, the BJKST algorithm uses a larger τ , namely, $\tau = 576/\epsilon^2$, such that the probability of a bad estimate can be bounded by $1/6$. Because $\frac{1}{6} + \frac{1}{6} = \frac{1}{3}$, a result akin to Eq. (1) can be established. Finally, taking the median over $k = 36 \log(1/\delta)$ instances results in an (ϵ, δ) -approximation.

The error guarantee, space bound, and time bound are summarized in the following theorem. The space bound includes the space for representing the hash functions. The time bound assumes that computing hash functions and b are constant expected time operations.

Theorem 6 ([2]) *The BJKST algorithm provides an (ϵ, δ) -approximation scheme, using $O((\frac{1}{\epsilon^2}(\log(1/\epsilon) + \log \log n) + \log n) \log(1/\delta))$ memory bits. For each item, the algorithm performs $O(\log(1/\delta))$ operations on $(\log_2 n)$ -bit words plus at most $O(\frac{\log(1/\delta)}{\epsilon^2})$ operations on $(\log_2(1/\epsilon) + \log_2 \log_2 n)$ -bit words.*

5 Lower Bounds

This section presents five key lower bound results for distinct-values estimation.

The first lower bound shows that *observing (nearly) the entire stream* is essential for obtaining good estimation error guarantees for all input streams.

Theorem 7 ([5]) *Consider any (possibly adaptive and randomized) estimator for the number of distinct values F_0 that examines at most r items in a stream of N items. Then, for any $\gamma > e^{-r}$, there exists a worst case input stream such that with probability at least γ , the ratio error of the estimate \hat{F}_0 output by the estimator is at least $\sqrt{\frac{N-r}{2r} \ln \frac{1}{\gamma}}$.*

Thus when $r = o(N)$, the ratio error is non-constant with high probability. Even when 1 % of the input is examined, the ratio error is at least 5 with probability $> 1/2$.

The second lower bound shows that *randomization* is essential for obtaining low estimation error guarantees for all input streams, if we hope to use sublinear space. For this lower bound, we also provide the proof, as a representative example of how such lower bounds are proved.

Theorem 8 ([1]) *Any deterministic algorithm that outputs, given one pass through a data stream of $N = n/2$ elements of $U = \{1, 2, \dots, n\}$, an estimate with at most 10 % relative error requires $\Omega(n)$ memory bits.*

Proof Let G be a family of $t = 2^{\Omega(n)}$ subsets of U , each of cardinality $n/4$ so that any two distinct members of G have at most $n/8$ elements in common. (The existence of such a G follows from standard results in coding theory, and can be proved by a simple counting argument.) Fix a deterministic algorithm that approximates F_0 . For every two members G_1 and G_2 of G let $A(G_1, G_2)$ be the stream of length $n/2$ starting with the $n/4$ members of G_1 (in a sorted order) and ending with the set of $n/4$ members of G_2 (in a sorted order). When the algorithm runs, given a stream of the form $A(G_1, G_2)$, the memory configuration after it reads the first $n/4$ elements of the stream depends only on G_1 . By the pigeonhole principle, if the memory has

less than $\log_2 t$ bits, then there are two distinct sets G_1 and G_2 in G , so that the content of the memory after reading the elements of G_1 is equal to that content after reading the elements of G_2 . This means that the algorithm must give the same final output to the two streams $A(G_1, G_1)$ and $A(G_2, G_1)$. This, however, contradicts the assumption, because $F_0 = n/4$ for $A(G_1, G_1)$ and $F_0 \geq 3n/8$ for $A(G_2, G_1)$. Therefore, the answer of the algorithm makes a relative error that exceeds 0.1 for at least one of these two streams. It follows that the space used by the algorithm must be at least $\log_2 t = \Omega(n)$, completing the proof. \square

The third lower bound shows that *approximation* is essential for obtaining low estimation error guarantees for all input streams, if we hope to use sublinear space.

Theorem 9 ([1]) *Any randomized algorithm that outputs, given one pass through a data stream of at most $N = 2n$ items of $U = \{1, 2, \dots, n\}$, a number Y such that $Y = F_0$ with probability at least $1 - \delta$, for some fixed $\delta < 1/2$, requires $\Omega(n)$ memory bits.*

The fourth lower bound shows that $\Omega(\log n)$ memory bits are required for obtaining low estimation error.

Theorem 10 ([1]) *Any randomized algorithm that outputs, given one pass through a data stream of items from $U = \{1, 2, \dots, n\}$, an estimate with at most a 10 % relative error with probability at least 3/4 must use at least $\Omega(\log n)$ memory bits.*

The final lower bound shows that $\Omega(1/\epsilon^2)$ memory bits are required in order to obtain an (ϵ, δ) -approximation scheme (even for constant δ).

Theorem 11 ([34]) *For any δ independent of n and any ϵ , any randomized algorithm that outputs, given one pass through a data stream of items from $U = \{1, 2, \dots, n\}$, an estimate with at most an ϵ relative error with probability at least $1 - \delta$ must use at least $\Omega(\min(n, 1/\epsilon^2))$ memory bits.*

Thus we have an $\Omega(1/\epsilon^2 + \log n)$ lower bound for obtaining arbitrary relative error for constant δ and, by the BJKST algorithm, a nearly matching upper bound of $O(1/\epsilon^2(\log(1/\epsilon) + \log \log n) + \log n)$.

6 Extensions

In this section, we consider distinct-values estimation in a variety of important scenarios beyond the basic data stream set-up. In Sects. 6.1–6.5, we focus on sampling, sliding windows, update streams, distributed streams, and sensor networks (ODI), respectively, as summarized in Table 2. Finally, Sect. 6.6 highlights three additional settings studied in the literature.

Table 2 Scenarios handled by the main algorithms discussed in this section

Algorithm	Cite & section	Sampling distinct	Sliding windows	Update streams	Distributed streams	ODI
FM	[13]; Sect. 3.1	no	no	no	yes	yes
PCSA	[13]; Sect. 3.4	no	no	no	yes	yes
FM with \log^2 space	Sects. 6.2, 6.3	no	yes	yes	yes	no
AMS	[1]; Sect. 3.2	no	no	no	yes	yes
Cohen	[7]; Sect. 3.3	no	no	no	yes	yes
LogLog	[11]; Sect. 3.4	no	no	no	yes	yes
Coordinated sampling	[17]; Sect. 4.1	yes	no	no	yes	yes
BJKST	[2]; Sect. 4.2	no	no	no	yes	yes
Distinct sampling	[16]; Sect. 6.1	yes	no	no	yes	yes
Randomized wave	[18]; Sect. 6.2	yes	yes	no	yes	yes
l_0 sketch	[9]; Sect. 6.2	no	no	yes	yes	no
Ganguly	[14]; Sect. 6.3	yes	no	yes	yes	no
CLKB	[8]; Sect. 6.5	no	no	no	yes	yes

6.1 Sampling Distinct

In addition to providing an estimate of the number of distinct values in the stream, several algorithms provide a *uniform sample of the distinct values* in the stream. Such a sample can be used for a variety of sampling-based estimation procedures, such as estimating the mean, the variance, and the quantiles over the distinct values. Algorithms that retain only hashed values, such as FM, PCSA, AMS, Cohen, LogLog, BJKST, l_0 Sketch (Sect. 6.2) and CLKB (Sect. 6.5), do not provide such samples. In some cases, such as Cohen, the algorithm can be readily adapted to produce a uniform sample (with replacement): For each instance (i.e., each hash function) of the algorithm, maintain not just the current minimum hashed value but also the original value associated with this minimum hashed value. As long as two different values do not hash to the same minimum value for a given hash function, each parallel instance produces one sample point. In contrast, Coordinated Sampling, Randomized Wave (Sect. 6.2) and Ganguly’s algorithm (Sect. 6.3) all directly provide a uniform sample of the distinct values.

Gibbons [16] extended the sampling goal to a multidimensional data setting that arises in a class of common databases queries. Here, the goal is to extract a uniform sample of the distinct values in a primary dimension, as before, but instead of retaining only the randomly selected values V , the algorithm retains a “same-value” sample for each value in V . Specifically, for each $v \in V$, the algorithm maintains a uniform random sample chosen from all the stream items with value v . A user-specified parameter t determines the size of each of these same-value samples; if there are fewer than t stream items with a particular value, the algorithm retains them all. The algorithm, called *Distinct Sampling*, is similar to Coordinated Sampling (Fig. 9) in having $\log n$ levels, maintaining all values whose levels are above

select count(distinct target-attr) from Table where P (a)	select count(distinct o_custkey) from orders where o_orderdate \geq ‘2006-01-01’ (b)
--	---

Fig. 10 (a) Distinct values query template; (b) Example query

a current threshold, and incrementing the level threshold whenever a space bound is reached. However, instead of retaining one (v, b) pair for the value v , it starts by retaining each of the first t items with value v in the primary dimension, as well as a count, n_v , of the number of items in the stream with value v (including the current item). Then, upon observing any subsequent items with value v , it maintains a uniform same-value sample for v by adding the new item to the sample with probability t/n_v , making room by discarding a random item among the t items currently in the sample for v . Figure 10 gives an example of the type of SQL query that can be well estimated by the Distinct Sampling algorithm, where target-attr in Fig. 10(a) is the primary dimension and the predicate P is typically on one or more of the other dimensions, as in Fig. 10(b). The estimate is obtained by first applying the predicate to the same-value samples, in order to estimate what fraction of the values in V would be eliminated by the predicate, and then outputting the overall query estimate based on the number of remaining values.

6.2 Sliding Windows

The sliding windows setting is motivated by the desire to estimate the number of distinct values over only the most recent stream items. Specifically, we are given a window size W , and the problem is to estimate the number of distinct values over a sliding window of the W most recent items. The goal is to use space that is logarithmic in W (linear in W would be trivial). Datar et al. [10] observed that the FM algorithm can be extended to solve the sliding windows problem, by keeping track of the stream position of the most recent item that set each FM bit. Then, when estimating the number of distinct values within the current sliding window, only those FM bits whose associated positions are within the window are considered to be set. This increases the space needed for the FM algorithm by a logarithmic factor.

Gibbons and Tirthapura [18] developed an (ϵ, δ) -approximation scheme for the sliding windows scenario. Their algorithm, called *Randomized Wave*, is depicted in Fig. 11. In the algorithm, there are $k = \Theta(\log(1/\delta))$ instances of the same procedure, differing only in their use of different hash functions $h_j()$. Any uniform, pairwise independent hash function can be used for $h_j()$. Let $b_j(v)$ be the value of b computed in the algorithm for $h_j(v)$.

Whereas Coordinated Sampling maintained a single uniform sample of the distinct values, Randomized Wave maintains $\approx \log W$ uniform samples of the distinct values. Each of these “level” samples corresponds to a different sampling probability, and retains only the $\tau = \Theta(1/\epsilon^2)$ most recent distinct values sampled into the

```

// Note: All additions and comparisons in this algorithm are done modulo  $W'$ 
 $W' :=$  the smallest power of 2 greater than or equal to  $2W$ 
 $pos := 0$ 
for  $j := 1, \dots, k$  do {
    initialize  $V_j$  to be an empty list // value list
    for  $\ell := 0, \dots, \log(W')$  do
        initialize  $L_j(\ell)$  to be an empty list // level lists
    }
// Process the stream items
foreach (stream item with value  $v$ ) do {
     $pos := pos + 1$ 
    for  $j := 1, \dots, k$  do {
        if the tail  $(v', p')$  of  $V_j$  has expired (i.e.,  $p' = pos - W$ )
            discard  $(v', p')$  from  $V_j$  and from any level list  $L_j()$ 
         $b :=$  the largest  $i \geq 0$  such that the  $i$  rightmost bits in  $h_j(v)$  are all 0
        for  $\ell := 0, \dots, b$  do {
            if  $v$  is already in  $L_j(\ell)$ 
                remove current entry for  $v$  and insert  $(v, pos)$  at the head of  $L_j(\ell)$ 
            else do {
                if  $|L_j(\ell)| = \tau$  then discard the pair at the tail of  $L_j(\ell)$ 
                insert  $(v, pos)$  at the head of  $L_j(\ell)$ 
            }
        }
        if  $v$  is in  $V_j$ 
            remove current entry for  $v$  from  $V_j$ 
        insert  $(v, pos)$  at the head of  $V_j$ 
    }
}
// Compute an estimate for a sliding window of size  $w \leq W$ 
 $s := \max(0, pos - w + 1)$  //  $[s, pos]$  is the desired window
for  $j := 1, \dots, k$  do {
     $\ell_j :=$  min level such that the tail of  $L_j(\ell_j)$  contains a position  $p \leq s$ 
     $c_j :=$  number of pairs in  $L_j(\ell_j)$  with  $p \geq s$ 
}
return median $j=1, \dots, k$ ( $c_j \cdot 2^{\ell_j}$ )

```

Fig. 11 The Randomized Wave algorithm [18]. The values of k and τ depend on the desired ϵ and δ : $k = 36 \log_2(1/\delta)$ and $\tau = 36/\epsilon^2$, where the constant 36 is determined by the worst case analysis and can be much smaller in practice

associated level. (In the figure, $L_j(\ell)$ is the level sample for level ℓ of instance j .) An item with value v is selected into levels $0, \dots, b_j(v)$, and stored as the pair (v, pos) , where pos is the stream position when v most recently occurred.

Each level sample $L_j(\ell)$ can be maintained as a doubly linked list. The algorithm also maintains a (doubly linked) list V_j of all the values in any of the level samples $L_j()$, together with the position of their most recent occurrences, ordered by position. This list enables fast discarding of items no longer within the sliding window. Finally, there is a hash table H_j (not shown in the figure) that holds triples $(v, Vptr, Lptr)$, where v is a value in V_j , $Vptr$ is a pointer to the entry for v in V_j ,

and $Lptr$ is a doubly linked list of pointers to each of the occurrences of v in the level samples $L_j()$. These triples are stored in H_j hashed by their value v .

Consider an instance j . For each stream item, we first check the oldest value v' in V_j to see if its position is now outside of the window, and if so, we discard it. We use the triple in $H_j(v')$ to locate all occurrences of v' in the data structures; these occurrences are spliced out of their respective doubly linked lists. Second, we update the level samples L_j for each of the levels $0 \dots b_j(v)$, where v is the value of the stream item. There are two cases. If v is not in the level sample, we insert it, along with its position pos , at the head of the level sample. Otherwise, we perform a move-to-front: splicing out v 's current entry and inserting (v, pos) at the head of the level sample. In the former case, if inserting the new element would make the level sample exceed τ elements, we discard the oldest element to make room. Finally, we insert (v, pos) at the head of V_j , and if v was already in V_j , we splice out the old entry for v . Because the expected value of $b_j(v)$ is less than 2, v occurs in an expected constant number (in this case, 2) of levels. Thus, all of the above operations can be done in constant expected time.

Let (v', p') denote the pair at the tail of a level sample $L_j(\ell)$. Then $L_j(\ell)$ contains all the distinct values with stream positions in the interval $[p', pos]$ whose $b_j()$'s are at least ℓ . Thus, similar to Coordinated Sampling, an estimate of the number of distinct values within a window can be obtained by taking the number of elements in $L_j(\ell)$ in this interval and multiplying by 2^ℓ , the inverse of the sampling probability for the level.

The error guarantee, space bound, and time bound are summarized in the following theorem. The space bound includes the space for representing the hash functions. The time bound assumes that computing hash functions and b are constant expected time operations.

Theorem 12 ([18]) *The Randomized Wave algorithm provides an (ϵ, δ) -approximation scheme for estimating the number of distinct values in any sliding window of size $w \leq W$, using $O(\frac{\log n \log W \log(1/\delta)}{\epsilon^2})$ memory bits, where the values are in $[0..n]$. For each item, the algorithm performs an expected $O(\log(1/\delta))$ operations on $\max(\log n, 2 + \log W)$ -bit memory words.*

Note that by setting W to be N , the length of the stream, the algorithm provides an (ϵ, δ) -approximation scheme for all possible window sizes.

6.3 Update Streams

Another important scenario is where the stream contains both new items and the deletion of previous items. Examples include estimating the current number of distinct network connections, phone connections or IP flows, where the stream contains both the start and the end of each connection or flow. Most of the distinct-values

algorithms discussed thus far are not designed to handle deletions. For example, algorithms that retain only the maximum of some quantity, such as AMS, Cohen and LogLog, or even the top few highest priority items, such as Coordinated Sampling and BJKST, are unable to properly account for the deletion of the current maximum or a high priority item. Similarly, once a bit i is set in the FM algorithm, the subsequent deletion of an item mapped to bit i does not mean the bit can be unset: there are likely to have been other un-deleted stream items that also mapped to bit i . In the case of FM, deletions can be handled by replacing each bit with a running counter that is incremented on insertions and decremented on deletions—at a cost of increasing the space needed by a logarithmic factor.

Update streams generalize the insertion and deletion scenario by having each stream item being a pair (v, Δ) , where $\Delta > 0$ specifies Δ insertions of the value v and $\Delta < 0$ specifies $|\Delta|$ deletions of the value v . The resulting frequency $f_v = \sum_{(v, \Delta) \in S} \Delta$ of value v is assumed to be nonnegative. The metric F_0 is the number of distinct values v with $f_v > 0$. The above variant of FM with counters instead of bits readily handles update streams. Cormode et al. [9] devised an (ϵ, δ) -approximation scheme, called l_0 sketch, for distinct-values estimation over update streams. Unlike any of the approaches discussed thus far, the l_0 sketch algorithm uses properties of *stable distributions*, and requires floating point arithmetic. The algorithm uses $O(\frac{1}{\epsilon^2} \log(1/\delta))$ floating point numbers and $O(\frac{1}{\epsilon^2} \log(1/\delta))$ floating point operations per stream item.

Recently, Ganguly [14] devised two (ϵ, δ) -approximation schemes for update streams. One uses $O(\frac{1}{\epsilon^2} (\log n + \log N) \log N \log(1/\delta))$ memory bits and $O(\log(1/\epsilon) \cdot \log(1/\delta))$ operations to process each stream update. The other uses a factor of $(\log(1/\epsilon) + \log(1/\delta))$ times more space but reduces the number of operations to only $O(\log(1/\epsilon) + \log(1/\delta))$. Both algorithms return a uniform sampling of the distinct values, as well as an estimate.

6.4 Distributed Streams

In a number of the motivating scenarios, the goal is to estimate the number of distinct values over a collection of *distributed streams*. For example, in network monitoring, each router observes a stream of packets and the goal is to estimate the number of distinct “values” (e.g., destination IP addresses, source–destination pairs, requested urls, etc.) across all the streams. Formally, we have $t \geq 2$ data streams, S_1, S_2, \dots, S_t , of items, where each item is from a universe of n possible values. Each stream S_i is observed and processed by a party, P_i , independently of the other streams, in one pass and with limited working space memory. The working space can be initialized (prior to observing any stream data) with data shared by all parties, so that, for example, all parties can use the same random hash function(s). The goal is to estimate the number of distinct values in the multi-set arising from concatenating all t streams. For example, in the $t = 2$ streams in Fig. 1, there are 15 distinct values in the two streams altogether.

In response to a request to produce an estimate, each party sends a message (containing its current synopsis or some function of it) to a Referee, who outputs the estimate based on these messages. Note that the parties do not communicate directly with one another, and the Referee does not directly observe any stream data. We are primarily interested in minimizing: (i) the workspace used by each party, and (ii) the time taken by a party to process a data item.

As shown in Table 2, each of the algorithms discussed in this chapter can be readily adapted to handle the distributed streams setting. For example, the FM algorithm (Fig. 3) can be applied to each stream independently, using the exact same hash function across all streams, to generate a bit vector $M[\cdot]$ for each stream. These bit vectors are sent to the Referee. Because the same hash function was used by all parties, the bit-wise OR of these t bit vectors yields exactly the bit vector that would have been produced by running the FM algorithm on the concatenation of the t streams (or any other interleaving of the stream data). Thus, the Referee computes this bit-wise OR, and then computes Z , the least significant 0-bit in the result. As in the original FM algorithm, we reduce the variance in the estimator by using k hash functions instead of just 1, where all parties use the same k hash functions. The Referee computes the Z corresponding to each hash function, then computes the average, \bar{Z} , of these Z 's, and finally, outputs $\lfloor 2^{\bar{Z}} / 0.77351 \rfloor$. The error guarantees of this distributed streams algorithm match the error guarantees in Theorem 1 for the single-stream algorithm. Moreover, the *per-party* space bound and the *per-item* time bound also match the space and time bounds in Theorem 1.

Similarly, PCSA, FM with \log^2 space, AMS, Cohen, and LogLog can be adapted to the distributed streams setting in a straightforward manner, preserving the error guarantees, per-party space bounds, and per-item time bounds of the single-stream algorithm.

A bit less obvious, but still relatively straightforward, is adapting algorithms that use dynamic thresholds on what to keep and what to discard, where the threshold adjusts to the locally-observed data distribution. The key observation for why these algorithms do not pose a problem is that we can match the error guarantees of the single-stream algorithm by having the Referee use the strictest threshold among all the local thresholds. (Here, “strictest” means that the smallest fraction of the data universe has its items kept.) Namely, if ℓ is the strictest threshold, the Referee “subsamples” the synopses from all the parties by applying the threshold ℓ to the synopses. This unifies all the synopses to the same threshold, and hence the Referee can safely combine these synopses and compute an estimate.

Consider, for example, the Coordinated Sampling algorithm (Fig. 9). Each party sends its sets S_1, \dots, S_k and levels ℓ_1, \dots, ℓ_k to the Referee. For $j = 1, \dots, k$, the Referee computes ℓ_j^* , the maximum value of the ℓ_j 's from all the parties. Then, for each j , the Referee subsamples each of the S_j from the t parties, by discarding from S_j all pairs (v', b') such that $b' < \ell_j^*$. Next, for each j , the Referee determines the union, S_j^* , of all the subsampled S_j 's. Finally, the Referee outputs the median over all j of $|S_j^*| \cdot 2^{l_j^*}$. The error guarantees, per-party space bound, and per-item time bound match those in Theorem 5 for the single-stream algorithm [17]. The error

guarantees follow because (i) S_j^* contains *all* pairs (v, b) with $b \geq \ell_j^*$ across all t streams, and (ii) the size of each S_j^* is at least as big as the size of the S_j at a party with level $\ell_j = \ell_j^*$ (i.e., at a party with no subsampling), and this latter size was already sufficient to get a good estimate in the single-stream setting.

6.5 Order- and Duplicate-Insensitive (*ODI*)

Another interesting setting for distinct-values estimation algorithms arises in robust aggregation in wireless sensor networks. In sensor network aggregation, an aggregate function (e.g., count, sum, average) of the sensors' readings is often computed by having the wireless sensor nodes organize themselves into a tree (with the base station as the root). The aggregate is computed bottom-up starting at the leaves of the tree: each internal node in the tree combines its own reading with the partial results from its children, and sends the result to its parent. For a sum, for example, the node's reading is added to the sum of its children's respective partial sums. This conserves energy because each sensor node sends only one short message, in contrast to the naive approach of having all readings forwarded hop-by-hop to the base station.

Aggregating along a tree, however, is not robust against message loss, which is common in sensor networks, because each dropped message loses a subtree's worth of readings. Thus, Considine et al. [8] and Nath et al. [24] proposed using multi-path routing for more robust aggregation. In one scheme, the nodes organize themselves into “rings” around the base station, where ring i consists of all nodes that are i hops from the base station. As in the tree, aggregation is done bottom-up starting with the nodes in the ring furthest from the base station (the “leaf” nodes). In contrast to the tree, however, when a node sends its partial result, there is no designated parent. Instead, all nodes in the next closest ring that overhear the partial result incorporate it into their accumulating partial results. Because of the added redundancy, the aggregation is highly robust to message loss, yet the energy consumption is similar to the (non-robust) tree because each sensor node sends only one short message.

On the other hand, because of the redundancy, partial results are accounted for multiple times. Thus, the aggregation must be done in a duplicate-insensitive fashion. This is where distinct-values estimation algorithms come in. First, if the goal is to count the number of distinct “values” (e.g., the number of distinct temperature readings), then a distinct-values estimation algorithm can be used, as long as the algorithm works for distributed streams and is insensitive to the duplication and observation re-ordering that arises in the scheme. An aggregation algorithm with the combined properties of order- and duplicate-insensitivity is called *ODI*-correct [24]. Second, if the goal is to count the number of sensor nodes whose readings satisfy a given boolean predicate (e.g., nodes with temperature readings below freezing), then again a distinct-values estimation algorithm can be used, as follows. Each sensor node whose reading satisfies the predicate uses its unique sensor id as its “value”. Then the number of distinct values in the sensor network is precisely the desired

count. Thus any distributed, ODI-correct distinct-values estimation algorithm can be used.

As shown in Table 2, most of the algorithms discussed in this chapter are ODI-correct. For example, the FM algorithm (Fig. 3) is insensitive to both re-ordering and duplication: the bits that are set in an FM bit vector are independent of both the order in which stream items are processed and any duplication of “partial-result” bit vectors (i.e., bit vectors corresponding to a subset of the stream items). Moreover, Considine et al. [8] showed how the FM algorithm can be effectively adapted to use only $O(\log \log n)$ bit messages in this setting. Similarly, most of the other algorithms are ODI-correct, as can be proved formally using the approach described in [24].

6.6 Additional Settings

We conclude this chapter by briefly mentioning three additional important settings considered in the literature.

The first setting is distinct-values estimation when each value is *unique*. This setting occurs, for example, in distributed census taking over mobile objects (e.g., [31]). Here, there are a large number of objects, each with a unique id. The goal is to estimate how many objects there are despite the constant motion of the objects, while minimizing the communication. Clearly, any of the distributed distinct-values estimation algorithms discussed in this chapter can be used. Note, however, that the setting enables a practical optimization: hash functions are not needed to map values to bit positions or levels. Instead, independent coin tosses can be used at each object; the desired exponential distribution can be obtained by flipping a fair coin until the first heads and counting the number of tails observed prior to the first head. (The unique id is not even used.) Obviating the need for hash functions eliminates their space and time overhead. Thus, for example, only $O(\log \log n)$ -bit synopses are needed for the AMS algorithm. Note that hash functions were needed before to ensure that the multiple occurrences of the same value all map to the same bit position or level; this feature is not needed in the setting with unique values.

A second setting, studied by Bar-Yossef et al. [3] and Pavan and Tirthapura [28], seeks to estimate the number of distinct values where each stream item is a *range* of integers. For example, in the 4-item stream [2, 5], [10, 12], [4, 8], [6, 7], there are 10 distinct values, namely, 2, 3, 4, 5, 6, 7, 8, 10, 11, and 12. Pavan and Tirthapura present an (ϵ, δ) -approximation scheme that uses $O(\frac{\log n \log(1/\delta)}{\epsilon^2})$ memory bits, and performs an amortized $O(\log(1/\delta) \log(n/\epsilon))$ operations per stream item. Note that although a single stream item introduces up to n distinct values into the stream, the space and time bounds have only a logarithmic (and not a linear) dependence on n .

Finally, a third important setting generalizes the distributed streams setting from just the union (concatenation) of the data streams to arbitrary set-expressions among the streams (including intersections and set differences). In this setting the number of distinct values corresponds to the *cardinality* of the resulting set. Ganguly et al. [15] showed how techniques for distinct values estimation can be generalized to handle this much richer setting.

References

1. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.* **58**, 137–147 (1999)
2. Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, Counting distinct elements in a data stream, in *Proc. 6th International Workshop on Randomization and Approximation Techniques* (2002), pp. 1–10
3. Z. Bar-Yossef, R. Kumar, D. Sivakumar, Reductions in streaming algorithms, with an application to counting triangles in graphs, in *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2002)
4. J. Bunge, M. Fitzpatrick, Estimating the number of species: a review. *J. Am. Stat. Assoc.* **88**, 364–373 (1993)
5. M. Charikar, S. Chaudhuri, R. Motwani, V. Narasayya, Towards estimation error guarantees for distinct values, in *Proc. 19th ACM Symp. on Principles of Database Systems* (2000), pp. 268–279
6. S. Chaudhuri, R. Motwani, V. Narasayya, Random sampling for histogram construction: how much is enough? in *Proc. ACM SIGMOD International Conf. on Management of Data* (1998), pp. 436–447
7. E. Cohen, Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.* **55**, 441–453 (1997)
8. J. Considine, F. Li, G. Kollios, J. Byers, Approximate aggregation techniques for sensor databases, in *Proc. 20th International Conf. on Data Engineering* (2004), pp. 449–460
9. G. Cormode, M. Datar, P. Indyk, S. Muthukrishnan, Comparing data streams using Hamming norms (how to zero in), in *Proc. 28th International Conf. on Very Large Data Bases* (2002), pp. 335–345
10. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows. *SIAM J. Comput.* **31**, 1794–1813 (2002)
11. M. Durand, P. Flajolet, Loglog counting of large cardinalities, in *Proc. 11th European Symp. on Algorithms* (2003), pp. 605–617
12. C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links, in *Proc. 3rd ACM SIGCOMM Conf. on Internet Measurement* (2003), pp. 153–166
13. P. Flajolet, G.N. Martin, Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.* **31**, 182–209 (1985)
14. S. Ganguly, Counting distinct items over update streams, in *Proc. 16th International Symp. on Algorithms and Computation* (2005), pp. 505–514
15. S. Ganguly, M. Garofalakis, R. Rastogi, Tracking set-expression cardinalities over continuous update streams. *VLDB J.* **13**, 354–369 (2004)
16. P.B. Gibbons, Distinct sampling for highly-accurate answers to distinct values queries and event reports, in *Proc. 27th International Conf. on Very Large Data Bases* (2001), pp. 541–550
17. P.B. Gibbons, S. Tirthapura, Estimating simple functions on the union of data streams, in *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures* (2001), pp. 281–291
18. P.B. Gibbons, S. Tirthapura, Distributed streams algorithms for sliding windows, in *Proc. 14th ACM Symp. on Parallel Algorithms and Architectures* (2002), pp. 63–72
19. P.J. Haas, J.F. Naughton, S. Seshadri, L. Stokes, Sampling-based estimation of the number of distinct values of an attribute, in *Proc. 21st International Conf. on Very Large Data Bases* (1995), pp. 311–322
20. P.J. Haas, L. Stokes, Estimating the number of classes in a finite population. *J. Am. Stat. Assoc.* **93**, 1475–1487 (1998)
21. W.C. Hou, G. Özsoyoglu, B.K. Taneja, Statistical estimators for relational algebra expressions, in *Proc. 7th ACM Symp. on Principles of Database Systems* (1988), pp. 276–287
22. W.C. Hou, G. Özsoyoglu, B.K. Taneja, Processing aggregate relational queries with hard time constraints, in *Proc. ACM SIGMOD International Conf. on Management of Data* (1989), pp. 68–77

23. A. Kumar, J. Xu, J. Wang, O. Spatscheck, L. Li, Space-code bloom filter for efficient per-flow traffic measurement, in *Proc. IEEE INFOCOM* (2004)
24. S. Nath, P.B. Gibbons, S. Seshan, Z. Anderson, Synopsis diffusion for robust aggregation in sensor networks, in *Proc. 2nd ACM International Conf. on Embedded Networked Sensor Systems* (2004), pp. 250–262
25. J.F. Naughton, S. Seshadri, On estimating the size of projections, in *Proc. 3rd International Conf. on Database Theory* (1990), pp. 499–513
26. F. Olken, Random sampling from databases. PhD thesis, Computer Science, UC, Berkeley (1993)
27. C.R. Palmer, P.B. Gibbons, C. Faloutsos, ANF: a fast and scalable tool for data mining in massive graphs, in *Proc. 8th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining* (2002), pp. 81–90
28. A. Pavan, S. Tirthapura, Range-efficient computation of F_0 over massive data streams, in *Proc. 21st IEEE International Conf. on Data Engineering* (2005), pp. 32–43
29. V. Poosala, Histogram-based estimation techniques in databases. PhD thesis, Univ. of Wisconsin-Madison (1997)
30. V. Poosala, Y.E. Ioannidis, P.J. Haas, E.J. Shekita, Improved histograms for selectivity estimation of range predicates, in *Proc. ACM SIGMOD International Conf. on Management of Data* (1996), pp. 294–305
31. Y. Tao, G. Kollios, J. Considine, F. Li, D. Papadias, Spatio-temporal aggregation using sketches, in *Proc. 20th International Conf. on Data Engineering* (2004), pp. 214–225
32. S. Venkataraman, D. Song, P.B. Gibbons, A. Blum, New streaming algorithms for high speed network monitoring and Internet attacks detection, in *Proc. 12th ISOC Network and Distributed Security Symp.* (2005)
33. K.Y. Whang, B.T. Vander-Zanden, H.M. Taylor, A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.* **15**, 208–229 (1990)
34. D. Woodruff, Optimal space lower bounds for all frequency moments, in *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms* (2004), pp. 167–175

The Sliding-Window Computation Model and Results

Mayur Datar and Rajeev Motwani

1 Sliding-Window Model: Motivation

In this chapter, we present some results related to small space computation over sliding windows in the data-stream model. Most research in the data-stream model (see, e.g., [1, 10, 11, 13–15, 19]), including results presented in some of the other chapters, assume that all data elements seen so far in the stream are equally important and synopses, statistics or models that are built should reflect the entire data set. However, for many applications this assumption is not true, particularly those that ascribe more importance to recent data items. One way to discount old data items and only consider recent ones for analysis is the *sliding-window model*: Data elements arrive at every instant; each data element expires after exactly N time steps; and, the portion of data that is relevant to gathering statistics or answering queries is the set of last N elements to arrive. The sliding window refers to the window of active data elements at a given time instant and window size refers to N .

1.1 Motivation and Road Map

Our aim is to develop algorithms for maintaining statistics and models that use space sublinear in the window size N . The following example motivates why we may not

M. Datar (✉)
Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA, USA
e-mail: mayur@google.com

R. Motwani
Department of Computer Science, Stanford University, Stanford, CA, USA
e-mail: rajeev@cs.stanford.edu

be ready to tolerate memory usage that is linear in the size of the window. Consider the following network-traffic engineering scenario: a high speed router working at 40 gigabits per second line speed. For every packet that flows through this router we do a prefix match to check if it originates from the `stanford.edu` domain. At every instant, we would like to know how many packets, of the last 10^{10} packets, belonged to the `stanford.edu` domain. The above question can be rephrased as the following simple problem:

Problem 1 (BASICCOUNTING) Given a stream of data elements, consisting of 0's and 1's, maintain at every time instant the count of the number of 1's in the last N elements.

A data element equals one if it corresponds to a packet from the `stanford.edu` domain and is zero otherwise. A trivial solution¹ exists for this problem that requires N bits of space. However, in such a scenario as the high-speed router, where on-chip memory is expensive and limited, and particularly when we would like to ask multiple (thousands) such continuous queries, it is prohibitive to use even $N = 10^{10}$ (window size) bits of memory for each query. Unfortunately, it is easy to see that the trivial solution is the best we can do in terms of memory usage, unless we are ready to settle for approximate answers, i.e., an exact solution to BASICCOUNTING requires $\Theta(N)$ bits of memory. We will present a solution to the problem that uses no more than $O(\frac{1}{\epsilon} \log^2 N)$ bits of memory (i.e., $O(\frac{1}{\epsilon} \log N)$ words of memory) and provides an answer at each instant that is accurate within a factor of $1 \pm \epsilon$. Thus, for $\epsilon = 0.1$ (10 % accuracy) our solution will use about 300 words of memory for a window size of 10^{10} .

Given our concern that derives from working with limited space, it is natural to ask "Is this the best we can do with respect with memory utilization?" We answer this question by demonstrating a matching space lower bound, i.e., we show that any approximation algorithm (deterministic or randomized) for BASICCOUNTING with relative error ϵ must use $\Omega(\frac{1}{\epsilon} \log^2 N)$ bits of memory. The lower bound proves that the above mentioned algorithm is optimal, to within constant factors, in terms of memory usage.

Besides maintaining simple statistics like a bit count, as in BASICCOUNTING, there are various applications where we would like to maintain more complex statistics. Consider the following motivating example:

A fundamental operation in database systems is a join between two or more relations. Analogously, one can define a join between multiple streams, which is primarily useful for correlating events across multiple data sources. However, since the input streams are unbounded, producing join results requires unbounded memory. Moreover, in most cases, we are only interested in those join results where the joining tuples exhibit temporal locality. Consequently, in most data-stream applications, a relevant notion of joins that is often employed is sliding-window joins, where tuples from each stream only join with tuples that belong to a sliding window over

¹Maintain a FIFO queue and update counter.

the other stream. The semantics of such a join are clear to the user and also such joins can be processed in a non-blocking manner using limited memory. As a result, sliding-window joins are quite popular in most stream applications.

In order to improve join processing, database systems maintain “join statistics” for the relations participating in the join. Similarly, in order to efficiently process sliding-window joins, we would like to maintain statistics over the sliding windows, for streams participating in the join. Besides being useful for the exact computation of sliding-window joins, such statistics could also be used to approximate them. Sliding-window join approximations have been studied by Das, Gehrke and Riedwald [6] and Kang, Naughton and Viglas [16]. This further motivates the need to maintain various statistics over sliding windows, using small space and update time.

This chapter presents a general technique, called the Exponential Histogram (EH) technique, that can be used to solve a wide variety of problems in the sliding-window model; typically problems that require us to maintain statistics. We will showcase this technique through solutions to two problems: the **BASICCOUNTING** problem above and the **SUM** problem that we will define shortly. However, our aim is not to solely present solutions to these problems, rather to explain the EH technique itself, such that the reader can appropriately modify it to solve more complex problems that may arise in various applications. Already, the technique has been applied to various other problems, of which we will present a summary in Sect. 5.

The road map for this chapter is as follows: After presenting an algorithm for the **BASICCOUNTING** problem and the associated space lower bound in Sects. 2 and 3, respectively, we present a modified version of the algorithm in Sect. 4 that solves the following generalization of the **BASICCOUNTING** problem:

Problem 2 (SUM) Given a stream of data elements that are positive integers in the range $[0..R]$, maintain at every time instant the sum of the last N elements.

A summary of other results in the sliding-window model is given in Sect. 5, before concluding in Sect. 6

2 A Solution to the **BASICCOUNTING** Problem

It is instructive to observe why naive schemes do not suffice for producing approximate answers with a low memory requirement. For instance, it is natural to consider random sampling as a solution technique for solving the problem. However, maintaining a uniform random sample of the window elements will result in poor accuracy in the case where the 1’s are relatively sparse.

Another approach is to maintain histograms. While the algorithm that we present follows this approach, it is important to note why previously known histogram techniques from databases are not effective for this problem. A histogram technique is characterized by the policy used to maintain the bucket boundaries. We would like to build time-based histograms in which every bucket summarizes a contiguous time

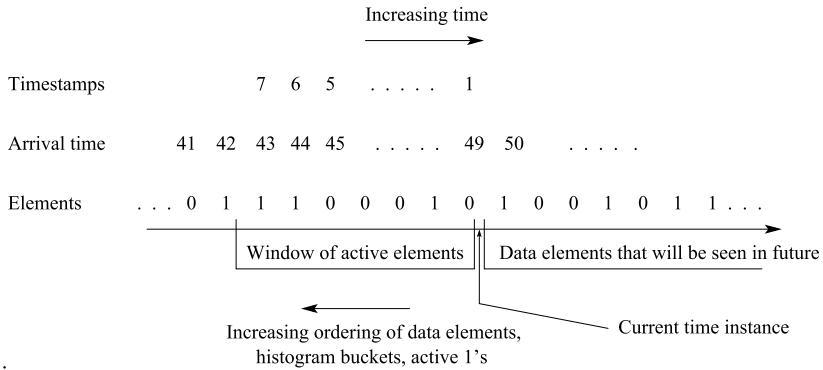


Fig. 1 Sliding window model notation

interval and stores the number of 1's that arrived in that interval. As with all histogram techniques, when a query is presented we may have to interpolate in some bucket to estimate the answer, because some of the bucket's elements may have expired. Let us consider some schemes of bucketizing and see why they will not work. The first scheme that we consider is that of dividing into k equi-width (width of time interval) buckets. The problem is that the distribution of 1's in the buckets may be nonuniform. We will incur large error when the interpolation takes place in buckets with a majority of the 1's. This observation suggests another scheme where we use buckets of nonuniform width, so as to ensure that each bucket has a near-uniform number of 1's. The problem is that total number of 1's in the sliding window could change dramatically with time, and current buckets may turn out to have more or less than their fair shares of 1's as the window slides forward. The solution we present is a form of histogram that avoids these problems by using a set of well-structured and nonuniform bucket sizes. It is called the Exponential Histogram (EH) for reasons that will be clear later. Before getting into the details of the solution we introduce some notation.

We follow the conventions illustrated in Fig. 1. In particular, we assume that new data elements are coming from the right and the elements at the left are ones already seen. Note that each data element has an *arrival time* which increments by one at each arrival, with the leftmost element considered to have arrived at time 1. But, in addition, we employ the notion of a *timestamp* which corresponds to the position of an *active* data element in the current window. We timestamp the active data elements from right to left, with the most recent element being at position 1. Clearly, the timestamps change with every new arrival and we do not wish to make explicit updates. A simple solution is to record the arrival times in a wraparound counter of $\log N$ bits and then the timestamp can be extracted by comparison with counter value of the current arrival. As mentioned earlier, we concentrate on the 1's in the data stream. When we refer to the k th 1, we mean the k th most recent 1 encountered in the data stream.

For an illustration of this notation, consider the situation presented in Fig. 1. The current time instant is 49 and the most recent arrival is a zero. The element with

arrival time 48 is the most recent 1 and has timestamp 2 since it is the second most recent arrival in the current window. The element with arrival time 44 is the second most recent 1 and has timestamp 6.

We will maintain histograms for the active 1's in the data stream. For every bucket in the histogram, we keep the timestamp of the most recent 1 (called *timestamp* for the bucket), and the number of 1's (called *bucket size*). For example, in our figure, a bucket with timestamp 2 and size 2 represents a bucket that contains the two most recent 1's with timestamps 2 and 6. Note that timestamp of a bucket increases as new elements arrive. When the timestamp of a bucket expires (reaches $N + 1$), we are no longer interested in data elements contained in it, so we drop that bucket and reclaim its memory. If a bucket is still active, we are guaranteed that it contains at least a single 1 that has not expired. Thus, at any instant there is at most one bucket (the last bucket) containing 1's that may have expired. At any time instant we may produce an estimate of the number of active 1's as follows. For all but the last bucket, we add the number of 1's that are in them. For the last bucket, let C be the count of the number of 1's in that bucket. The actual number of active 1's in this bucket could be anywhere between 1 and C , so we estimate it to be $C/2$. We obtain the following:

Fact 1 The absolute error in our estimate is at most $C/2$, where C is the size of the last bucket.

Note that, for this approach, the window size does not have to be fixed a-priori at N . Given a window size S ($S \leq N$), we do the same thing as before except that the last bucket is the bucket with the largest timestamp less than S .

2.1 The Approximation Scheme

We now define the Exponential Histograms and present a technique to maintain them, so as to guarantee count estimates with relative error at most ϵ , for any $\epsilon > 0$. Define $k = \lceil \frac{1}{\epsilon} \rceil$, and assume that $\frac{k}{2}$ is an integer; if $\frac{k}{2}$ is not an integer, we can replace $\frac{k}{2}$ by $\lceil \frac{k}{2} \rceil$ without affecting the basic results.

As per Fact 1, the absolute error in the estimate is $C/2$, where C is the size of the last bucket. Let the buckets be numbered from right to left with the most recent bucket being numbered 1. Let m denote the number of buckets and C_i denote the size of the i th bucket. We know that the true count is at least $1 + \sum_{i=1}^{m-1} C_i$, since the last bucket contains at least one unexpired 1 and the remaining buckets contribute exactly their size to total count. Thus, the relative estimation error is at most $(C_m/2)/(1 + \sum_{i=1}^{m-1} C_i)$. We will ensure that the relative error is at most $1/k$ by maintaining the following invariant:

Invariant 1 At all times, the bucket sizes C_1, \dots, C_m are such that for all $j \leq m$, we have $C_j/(2(1 + \sum_{i=1}^{j-1} C_i)) \leq \frac{1}{k}$.

Let $N' \leq N$ be the number of 1's that are active at any instant. Then the bucket sizes must satisfy $\sum_{i=1}^m C_i \geq N'$. Our goal is to satisfy this property and Invariant 1 with as few buckets as possible. In order to achieve this goal we maintain buckets with exponentially increasing sizes so as to satisfy the following second invariant.

Invariant 2 At all times the bucket sizes are nondecreasing, i.e., $C_1 \leq C_2 \leq \dots \leq C_{m-1} \leq C_m$. Further, bucket sizes are constrained to the following: $\{1, 2, 4, \dots, 2^{m'}\}$, for some $m' \leq m$ and $m' \leq \log \frac{2N}{k} + 1$. For every bucket size other than the size of the first and last bucket, there are at most $\frac{k}{2} + 1$ and at least $\frac{k}{2}$ buckets of that size. For the size of the first bucket, which is equal to one, there are at most $k + 1$ and at least k buckets of that size. There are at most $\frac{k}{2}$ buckets with size equal to the size of the last bucket.

Let $C_j = 2^r$ ($r > 0$) be the size of the j th bucket. If the size of the last bucket is 1 then there is no error in estimation since there is only data element in that bucket for which we know the timestamp exactly. If Invariant 2 is satisfied, then we are guaranteed that there are at least $\frac{k}{2}$ buckets each of sizes $2, 4, \dots, 2^{r-1}$ and at least k buckets of size 1, which have indexes less than j . Consequently, $C_j < \frac{2}{k}(1 + \sum_{i=1}^{j-1} C_i)$. It follows that if Invariant 2 is satisfied then Invariant 1 is automatically satisfied, at least with respect to buckets that have sizes greater than 1. If we maintain Invariant 2, it is easy to see that to cover all the active 1's, we would require no more than $m \leq (\frac{k}{2} + 1)(\log(\frac{2N}{k}) + 2)$ buckets. Associated with each bucket is its size and a timestamp. The bucket size takes at most $\log N$ values, and hence we can maintain them using $\log \log N$ bits. Since a timestamp requires $\log N$ bits, the total memory requirement of each bucket is $\log N + \log \log N$ bits. Therefore, the total memory requirement (in bits) for an EH is $O(\frac{1}{\epsilon} \log^2 N)$. It is implied that by maintaining Invariant 2, we are guaranteed the desired relative error and memory bounds.

The query time for EH can be made $O(1)$ by maintaining two counters, one for the size of the last bucket (LAST) and one for the sum of the sizes of all buckets (TOTAL). The estimate itself is TOTAL minus half of LAST. Both counters can be updated in $O(1)$ time for every data element. See the box below for a detailed description of the update algorithm.

Example 1 We illustrate the execution of the algorithm for 10 steps, where at each step the new data element is 1. The numbers indicate the bucket sizes from left to right, and we assume that $\frac{k}{2} = 1$.

```

32, 32, 16, 8, 8, 4, 2, 1, 1
32, 32, 16, 8, 8, 4, 4, 2, 1, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 1, 1, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1, 1, 1 (merged the older 1's)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1, 1, 1 (merged the older 1's)
32, 32, 16, 8, 8, 4, 4, 2, 1, 1, 1 (merged the older 2's)
32, 32, 16, 8, 8, 4, 2, 1, 1, 1 (merged the older 4's)
32, 32, 16, 8, 4, 2, 1, 1 (merged the older 8's)

```

Algorithm (Insert):

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket and update the counter LAST containing the size of the last bucket and the counter TOTAL containing the total size of the buckets.
2. If the new data element is 0 ignore it; else, create a new bucket with size 1 and the current timestamp, and increment the counter TOTAL.
3. Traverse the list of buckets in order of increasing sizes. If there are $\frac{k}{2} + 2$ buckets of the same size ($k + 2$ buckets if the bucket size equals 1), merge the oldest two of these buckets into a single bucket of double the size. (A merger of buckets of size 2^r may cause the number of buckets of size 2^{r+1} to exceed $\frac{k}{2} + 1$, leading to a cascade of such mergers.) Update the counter LAST if the last bucket is the result of a new merger.

Merging two buckets corresponds to creating a new bucket whose size is equal to the sum of the sizes of the two buckets and whose timestamp is the timestamp of the more recent of the two buckets, i.e., the timestamp of the bucket that is to the right. A merger requires $O(1)$ time. Moreover, while cascading may require $\Theta(\log \frac{2N}{k})$ mergers upon the arrival of a single new element, a simple argument, presented in the next proof, allows us to argue that the amortized cost of mergers is $O(1)$ per new data element. It is easy to see that the above algorithm maintains Invariant 2. We obtain the following theorem:

Theorem 1 *The EH algorithm maintains a data structure that gives an estimate for the BASICCOUNTING problem with relative error at most ϵ using at most $(\frac{k}{2} + 1)(\log(\frac{2N}{k}) + 2)$ buckets, where $k = \lceil \frac{1}{\epsilon} \rceil$. The memory requirement is $\log N + \log \log N$ bits per bucket. The arrival of each new element can be processed in $O(1)$ amortized time and $O(\log N)$ worst-case time. At each time instant, the data structure provides a count estimate in $O(1)$ time.*

Proof The EH algorithm above, by its very design, maintains Invariant 2. As noted earlier, an algorithm that maintains Invariant 2, requires no more than $(\frac{k}{2} + 1)(\log(\frac{2N}{k}) + 2)$ buckets to cover all the active 1's. Furthermore, the invariant also guarantees that our estimation procedure has a relative error no more than $1/k \leq \epsilon$.

Each bucket maintains a timestamp and the size for that bucket. Since we maintain timestamps using wraparound arrival times, they require no more than $\log N$ bits of memory. As per Invariant 2, bucket sizes can take only one of the $\log \frac{2N}{k} + 1$ unique values, and can be represented using $\log \log N$ bits. Thus, the total memory requirement of each bucket is no more than $\log N + \log \log N$ bits.

On the arrival of a new element, we may perform a cascading merge of buckets, that takes time proportional to the number of buckets. Since there are $O(\log N)$ buckets, this gives a worst case update time of $O(\log N)$. Whenever two buckets are merged, the size of the merged bucket is double the size of those that are merged.

The cost of the merging can be amortized among all the 1's that fall in the merged bucket. Thus, an element that belongs to a bucket of size 2^p , pays an amortized cost $1 + 1/2 + 1/4 + \dots + 1/2^p \leq 2$. This is because, whenever it gets charged, the size of the bucket it belongs to doubles and consequently the charge it incurs halves. Thus, we get that the amortized cost of merging buckets is $O(1)$ per new element, in fact, $O(1)$ per new element that has value 1.

We maintain counters TOTAL and LAST, which can be updated in $O(1)$ time for each new element, and which enable us to give a count estimate in $O(1)$ time whenever a query is asked. \square

If instead of maintaining a timestamp for every bucket, we maintain a timestamp for the most recent bucket and maintain the difference between the timestamps for the successive buckets then we can reduce the total memory requirement to $O(k \log^2 \frac{N}{k})$.

3 Space Lower Bound for BASICCOUNTING Problem

We provide a lower bound which verifies that the algorithms is optimal in its memory requirement. We start with a deterministic lower bound of $\Omega(k \log^2 \frac{N}{k})$. We omit proofs for lack of space, and refer the reader to [8].

Theorem 2 *Any deterministic algorithm that provides an estimate for the BASICCOUNTING problem at every time instant with relative error less than $\frac{1}{k}$ for some integer $k \leq 4\sqrt{N}$ requires at least $\frac{k}{16} \log^2 \frac{N}{k}$ bits of memory.*

The proof argument goes as follows: At any time instant, the space utilized by any algorithm, is used to summarize the contents of the current active window. For a window of size N , we can show that there are a large number of possible input instances, i.e., arrangements of 0's and 1's, such that any deterministic algorithm which provides estimates with small relative error (i.e., less than $\frac{1}{k}$) has to differentiate between every pair of these arrangements. The number of memory bits required by such an algorithm must therefore exceed the logarithm of the number of arrangements. The above argument is formalized by the following lemma.

Lemma 1 *For $k/4 \leq B \leq N$, there exist $L = \binom{B}{k/4}^{\lfloor \log \frac{N}{B} \rfloor}$ arrangements of 0's and 1's of length N such that any deterministic algorithm for BASICCOUNTING with relative error less than $\frac{1}{k}$ must differentiate between any two of the arrangements.*

To prove Theorem 2, observe that if we choose $B = \sqrt{Nk}$ in the lemma above then $\log L \geq \frac{k}{16} \log^2 \frac{N}{k}$. While the lower bound above is for a deterministic algorithm, a standard technique for establishing lower bounds for randomized algorithms, called the *minimax principle* [18], lets us extend this lower bound on the space complexity to randomized algorithms.

As a reminder, a *Las Vegas algorithm* is a randomized algorithm that always produces the correct answer, although the running time or space requirement of the algorithm may vary with the different random choices that the algorithm makes. On the other hand, a *Monte Carlo algorithm* is a randomized algorithm that sometimes produces an incorrect solution. We obtain the following lower bounds for these two classes of algorithms.

Theorem 3 Any randomized Las Vegas algorithm for **BASICCOUNTING** with relative error less than $\frac{1}{k}$, for some integer $k \leq 4\sqrt{N}$, requires an expected memory of at least $\frac{k}{16} \log^2 \frac{N}{k}$ bits.

Theorem 4 Any randomized Monte Carlo algorithm for **BASICCOUNTING** problem with relative error less than $\frac{1}{k}$, for some integer $k \leq 4\sqrt{N}$, with probability at least $1 - \delta$ (for $\delta < \frac{1}{2}$) requires an expected memory of at least $\frac{k}{32} \log^2 \frac{N}{k} + \frac{1}{2} \log(1 - 2\delta)$ bits.

4 Beyond 0's and 1's

The **BASICCOUNTING** problem, discussed in the last two sections, is one of the basic operations that one can define over sliding windows. While the problem in its original form has various applications, it is natural to ask “What are the other problems, in the sliding-window model, that can be solved using small space and small update time?”. For instance, instead of the data elements being binary values, namely 0 and 1, what if they were positive integers in the range $[0..R]$? Could we efficiently maintain the sum of these numbers in the sliding-window model? We have already defined this problem, in Sect. 1, as the **SUM** problem.

We will now present a modification of the algorithm from Sect. 2, that solves the **SUM** problem. In doing so, we intend to highlight the characteristic elements of the solution technique, so that readers may find it easy to adapt the technique to other problems. Already, the underlying technique has been successfully applied to many problems, some of which will be listed in the following section.

One way to solve the **SUM** problem would be to maintain separately a sliding window sum for each of the $\log R$ bit positions using an EH from Sect. 2.1. As before, let $k = \lceil \frac{1}{\epsilon} \rceil$. The memory requirement for this approach would be $O(k \log^2 N \log R)$ bits. We will present a more direct approach that uses less memory. In the process we demonstrate how the EH technique introduced in Sect. 2 can be generalized to solving a bigger class of problems.

Typically, a problem in the sliding-window model requires us to maintain a function f defined over the elements in the sliding window. Let $f(B)$ denote the function value restricted to the elements in a bucket B . For example, in case of the **SUM** problem, the function f equals the sum of the positive integers that fall inside the sliding-window. In case of **BASIC COUNTING** the function f is simply the number of 1's that fall inside the sliding-window. We note the following central ingredients of the EH technique from Sect. 2 and adapt them for the **SUM** problem:

1. **Size of a Bucket.** The *size* of each bucket is defined as the value of the function f that we are estimating (over the sliding window), restricted to that bucket B , i.e., $f(B)$. In the earlier case *size* was simply the count of 1's falling inside the bucket. For SUM, we define *size* analogously as the sum of integers falling inside the bucket.
2. **Procedure to Merge Buckets or Combination Rule.** Whenever the algorithm decides to merge two adjacent buckets, a new bucket is created with timestamp equal to that of the newer bucket or the bucket to the right. The *size* of this new bucket is computed using the sizes of the individual buckets (i.e., using $f(B)$'s for the buckets that are merged) and any additional information that may be stored with the buckets.² Clearly, for the problem of maintaining the sum of data elements, which are either 0's and 1's or positive integers, no additional information is required. By definition, the size of the new merged bucket is simply the sum of the sizes of buckets being merged.
3. **Estimation.** Whenever a query is asked, we need to estimate the answer at that moment based on the sizes of all the buckets and any additional information that we may have kept. In order to estimate the answer, we may be required to “interpolate” over the last bucket that is part inside and part outside the sliding window, i.e., the “straddling” bucket.

Typically, this is done by computing the function value f over all buckets other than the last bucket. In order to do this, we use the same procedure as in the Merge step. To this value we may add the interpolated value of the function f from the last bucket.

Again, for the problem of maintaining the sum of positive integers this task is relatively straightforward. We simply add up the sizes of all the buckets that are completely inside the sliding window. To this we add the “interpolated” value from the last bucket, which is simply half the size of the last bucket.

4. **Deleting the Oldest Bucket.** In order to reclaim memory, the algorithm deletes the oldest bucket when its timestamp reaches $N + 1$. This step is same irrespective of the function f we are estimating.

The technique differs for different problems in the particulars of how the steps above are executed and the rules for when to merge old buckets and create new ones, as new data elements get inserted. The goal is to maintain as few buckets as possible, i.e., merge buckets whenever possible, while at the same time making sure that the error due to the estimation procedure, which interpolates for the last bucket, is bounded. Typically, this goal is achieved by maintaining that the bucket sizes grow exponentially from right to left (new to old) and hence the name Exponential Histograms (EH). It is shown in [8] that the technique can be used to estimate a general class of functions f , called *weakly-additive* functions, over sliding windows. In

²There are problems for which just knowing the sizes of the buckets that are merged is not sufficient to compute the size of the new merged bucket. For example, if the function f is the variance of numbers, in addition to knowing the variance of the buckets that are merged, we also need to know the number of elements in each bucket and mean value of the elements from each bucket, in order to compute the variance for the merged bucket; see [4] for details.

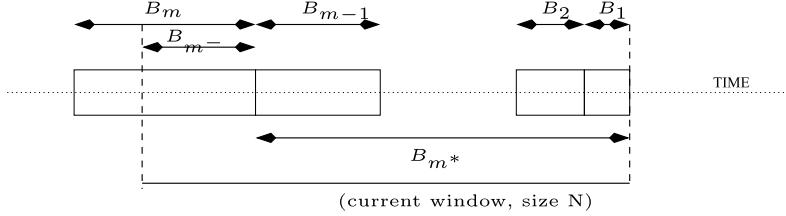


Fig. 2 An illustration of an Exponential Histogram (EH)

the following section, we list different problems over sliding windows, that can be solved using the EH technique.

We need some more notation to demonstrate the EH technique for the **SUM** problem. Let the buckets in the histogram be numbered \$B_1, B_2, \dots, B_m\$, starting from most recent (\$B_1\$) to oldest (\$B_m\$); further, \$t_1, t_2, \dots, t_m\$ denote the bucket timestamps; see Fig. 2 for an illustration. In addition to the buckets maintained by the algorithm, we define another set of *suffix buckets*, denoted \$B_{1*}, \dots, B_{j*}\$, that represent suffixes of the data stream. Bucket \$B_{i*}\$ represents all elements in the data stream that arrived after the elements of bucket \$B_i\$, that is, \$B_{i*} = \bigcup_{l=1}^{i-1} B_l\$. We do not explicitly maintain the suffix buckets. Let \$S_i\$ denote the size of bucket \$B_i\$. Similarly, let \$S_{i*}\$ denote the size of the suffix bucket \$B_{i*}\$. Note, for the **SUM** problem \$S_{i*} = \sum_{l=1}^{i-1} S_l\$. Let \$B_{i,i-1}\$ denote the bucket that would be formed by merging buckets \$i\$ and \$i-1\$, and \$S_{i,i-1}\$ (\$S_{i,i-1} = S_i + S_{i-1}\$) denote the size of this bucket. We maintain the following two invariants that guarantee a small relative error \$\epsilon\$ in estimation and small number of buckets:

Invariant 3 For every bucket \$B_i\$, \$\frac{1}{2\epsilon} S_i \leq S_{i*}\$.

Invariant 4 For each \$i > 1\$, for every bucket \$B_i\$,

$$\frac{1}{2\epsilon} S_{i,i-1} > S_{i-1*}.$$

It follows from Invariant 3 that the relative error in estimation is no more than \$\frac{S_m}{2S_{m*}} \leq \epsilon\$. Invariant 4 guarantees that the number of buckets is no more than \$O(\frac{1}{\epsilon}(\log N + \log R))\$. It is easy to see the proof of this claim. Since \$S_{i*} = \sum_{l=1}^{i-1} S_l\$, i.e., the bucket sizes are additive, after every \$\lceil 1/\epsilon \rceil\$ buckets (rather \$\lceil 1/2\epsilon \rceil\$ pairs of buckets) the value of \$S_{i*}\$ doubles. As a result, after \$O(\frac{1}{\epsilon}(\log N + \log R))\$ buckets the value of \$S_{i*}\$ exceeds \$NR\$, which is the maximum value that can be achieved by \$S_{i*}\$. We now present a simple insert algorithm that maintains the two invariants above as new elements arrive.

Note, Invariant 3 holds for buckets that have been formed as a result of the merging of two or more buckets because the merging condition assures that it holds for the merged bucket. Addition of new elements in the future does not violate the invariant, since the right-hand side of the invariant can only increase by addition of the

Algorithm (Insert): x_t denotes the most recent element.

1. If $x_t = 0$, then do nothing. Otherwise, create a new bucket for x_t . The new bucket becomes B_1 with $S_1 = x_t$. Every old bucket B_i becomes B_{i+1} .
2. If the oldest bucket B_m has timestamp greater than N , delete the bucket. Bucket B_{m-1} becomes the new oldest bucket.
3. **Merge step.** Let $k = \frac{1}{2\epsilon}$. While there exists an index $i > 2$ such that $kS_{i,i-1} \leq S_{i-1}^*$, find the smallest such i and combine buckets B_i and B_{i-1} using the combination rule described earlier. Note that S_{i-1}^* value can be computed incrementally by adding S_{i-1} and S_{i-1}^* , as we make the sweep.

new elements. However, the invariant may not hold for a bucket that contains a singleton nonzero element and was never merged. The fact that the invariant does not hold for such a bucket, does not affect the error bound for the estimation procedure because, if such a bucket were to become the last bucket, we know the exact timestamp for the only non zero element in the bucket. As a result there is no interpolation error in that case.

Analogously to the variables **TOTAL** and **LAST** in Sect. 2.1, we can maintain $S_m + S_{m^*}$ and S_m that enable us to answer queries in $O(1)$ time. The algorithm for insertion requires $O(\frac{1}{\epsilon}(\log N + \log R))$ time per new element. Most of the time is spent in Step 3, where we make the sweep to combine buckets. This time is proportional to number of buckets, $(O(\frac{1}{\epsilon}(\log N + \log R)))$. A simple trick, to skip Step 3 until we have seen $\Theta(\frac{1}{\epsilon}(\log N + \log R))$ data points, ensures that the running time of the algorithm is amortized $O(1)$. While we may violate Invariant 4 temporarily, we restore it after seeing $\Theta(\frac{1}{\epsilon}(\log N + \log R))$ data points, by executing Step 3, which ensures that the number of buckets is $O(\frac{1}{\epsilon}(\log N + \log R))$. The space requirement for each bucket (memory needed to maintain timestamp and size) is $\log N + \log R$ bits. If we assume that a word is at least $\log N + \log R$ bits long, equivalently the size required to count up to NR , which is the maximum value of the answer, we get that the total memory requirement is $O(\frac{1}{\epsilon}(\log N + \log R))$ words or $O(\frac{1}{\epsilon}(\log N + \log R)^2)$ bits. Please refer to [8] for a more complex procedure that has similar time requirements and space requirement $O(\frac{1}{\epsilon} \log N (\log N + \log R))$ bits. To summarize, we get the following theorem:

Theorem 5 *The sum of positive integers in the range $[0..R]$ can be estimated over sliding windows with relative error at most ϵ using $O(\frac{1}{\epsilon}(\log N + \log R))$ words of memory. The time to update the underlying EH is worst case $O(\frac{1}{\epsilon}(\log N + \log R))$ and amortized $O(1)$.*

Similar to the space lower bound that we presented in Sect. 3, one can show a space lower bound of $\Omega(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ bits for the SUM problem; see [8] for details. This is asymptotically equal to the upper bound for the algorithm in [8] that we mentioned earlier.

It is natural to ask the question: What happens if we do not restrict data elements to positive integers and are interested in estimating the sum over sliding windows. We show that even if we restrict the set of unique data elements to $\{1, 0, -1\}$, to approximate the sum within a constant factor requires $\Omega(N)$ bits of memory. Moreover, it is easy to maintain the sum by storing the last N integers which requires $O(N)$ bits of memory. We assume that the storage required for every integer is a constant independent of the window size N . With this assumption, we have that the complexity of the problem in the general case (allowing positive and negative integers) is $\Theta(N)$.

We now argue the lower bound of $\Omega(N)$. Consider an algorithm A that provides a constant-factor approximation to the problem of maintaining the general sum. Given a bit vector of size $N/2$ we present the algorithm A with the pair $(-1, 1)$ for every 1 in the bit vector and the pair $(1, -1)$ for every 0. Consider the state (time instance) after we have presented all the $N/2$ pairs to the algorithm. We claim that we can completely recover the original bit vector by presenting a sequence of 0's henceforth and querying the algorithm on every odd time instance. If the current time instance is T (after having presented the $N/2$ pairs) then it is easy to see that the correct answer at time instance $T + 2i - 1$ ($1 \leq i \leq N/2$) is 1 iff the i th bit was 1 and -1 iff the i th bit was 0. Since the algorithm A gives a constant factor approximation its estimate would be positive if the correct answer is 1 and negative if the correct answer was -1 . Since the state of the algorithm after feeding the $N/2$ pairs enables us to recover the bit vector exactly for any arbitrary bit vector it must be using at least $N/2$ bits of memory to encode it. This proves the lower bound. We can state the following theorem:

Theorem 6 *The space complexity of any algorithm that gives a constant factor approximation, at every instant, to the problem of maintaining the sum of last N integers (positive or negative) that appear as stream of data elements is equal to $\Theta(N)$.*

5 References and Related Work

The EH technique that we demonstrate through solutions to the BASICCOUNTING and SUM problem is by Datar, Gionis, Indyk and Motwani [8]. The space lower bounds presented above are also from that paper. In the same paper, the authors characterize a general class of *weakly additive* functions that can be efficiently estimated over sliding windows, using the EH technique. Also see, Datar's PhD thesis [7] for more details.

As we have seen in other chapters from this book, it is often the case that input data streams are best visualized as a high dimensional vector. A standard operation

is to compute the l_p norm, for $0 < p \leq 2$, of these vectors or the l_p norm of the difference between two vectors. In Chap., we have seen *sketching* techniques to estimate these l_p norms using small space. It turns out that, when each data element in the data stream represents an increment to some dimension of the underlying high dimensional vector, the l_p norm of a vector belongs to the class of *weakly additive* functions mentioned above. Consequently, for the restricted case when the increments are positive, the EH technique in conjunction with the sketching technique, can be adapted to the estimate l_p norms over the sliding windows; see [7, 8] for details.

Babcock, Datar, Motwani and O'Callaghan [4] showed that the variance of real numbers with maximum absolute value R can be estimated over sliding windows with relative error at most ϵ using $O(\frac{1}{\epsilon^2}(\log N + \log R))$ words of memory. The update time for the data structure is worst case $O(\frac{1}{\epsilon^2}(\log N + \log R))$ and amortized $O(1)$. In the same paper, the authors look at the problem of maintaining k -medians clustering of points over a sliding window. They present an algorithm that uses $O(\frac{k}{\tau^4}N^{2\tau} \log^2 N)$ memory³ and presents k centers, for which the objective function value is within a constant factor ($2^{O(1/\tau)}$) of optimal, where $\tau < 1/2$ is a parameter which captures the trade-off between the space bound and the approximation ratio. The update time for the data structure is worst case $\tilde{O}(\frac{k^2}{\tau^3}N^{2\tau})$ and amortized $\tilde{O}(k)$. Both these algorithms are an adaptation of the EH technique, presented in Sect. 4 above.

In this chapter, we have focussed on the sliding-window model, that assumes that the pertinent data set is the last N data elements, i.e., we focus on *sequence-based* sliding-window model. In other words, we assumed that data items arrive at regular time intervals and arrival time increases by one with every new data item that we have seen. Such regularity in arrival of data items is seldom true for most real life applications, for which arrival rates of data items may be bursty. Often, we would like to define the sliding window based on real time. It is easy to adapt the EH technique to such a *time-based* sliding-window model; see [7, 8] for details.

One may argue that the sliding-window model is not the right model to discount old data, in the least not the only model. If our aim is to assign a smaller weight to older elements so that they contribute less to any statistics or models we maintain, we may want to consider other monotonically decreasing functions (time decayed functions) for assigning weights to elements other than the step function (1 for the last N elements and 0 beyond) that is implicit in the sliding-window model. A natural decay function is the exponentially decreasing weight function that was considered by Gilbert et al. [12] in maintaining *aged aggregates*: For a data stream $\dots, x_{(-2)}, x_{(-1)}, x_{(0)}$, where $x_{(0)}$ is the most recently seen data element, λ -aging aggregate is defined as $\lambda x_{(0)} + \lambda(1 - \lambda)x_{(-1)} + \lambda(1 - \lambda)^2x_{(-2)} + \dots$. Exponentially decayed statistics as above are easy to maintain, although one may argue that exponential decay of weights is not suited for all applications or is too restrictive.

³The space required to hold a single data point, which in this case is a point from some metric space, is assumed to be $O(1)$ words.

We may desire a richer class of decay functions, e.g., polynomially decaying weight functions instead of exponential decay. Cohen and Strauss [5] show how to maintain statistics efficiently for a general class of time decaying functions. Their solutions use the EH technique as a building block or subroutine, thereby demonstrating the applicability of the EH technique to a wider class of models that allow for time decay, besides the sliding-window model that we have considered.

See [7] for solutions to other problems in the sliding-window model that do not rely on the EH technique. These problems include maintaining a uniform random sample (see also [3]), maintaining the min/max of real numbers, estimating the ratio of *rare*⁴ elements to the number of distinct elements (see also [9]), and estimating the similarity between two data streams measured according to the Jaccard coefficient for set similarity between two sets $A, B: |A \cap B|/|A \cup B|$ (see also [9]).

Maintaining approximate counts of high frequency elements and maintaining approximate quantiles, are important problems that have been studied in database research as maintaining end-biased histograms and maintaining equi-depth histograms. These problems are particularly useful for sliding-window join processing; they provide the necessary join statistics and can also be used for approximate computation of joins. A solution to these problems, in the sliding-window model, is presented by Arasu and Manku [2] and Lu et al. [17].

6 Conclusion

In this chapter, we have studied algorithms for two simple problems, **BASICCOUNTING** and **SUM**, in the sliding-window model; a natural model to discount stale data that only considers the last N elements as being pertinent. Our aim was to showcase the Exponential Histogram (EH) technique that has been used to efficiently solve various problems over sliding windows. We also presented space lower bounds for the two problems above. See Table 1 for a summary of results in the sliding-window model. Note that, for this summary, we measure memory in words, where a word is assumed large enough to hold the answer or one unit of answer. For example, in the case of **BASICCOUNTING** a word is assumed to be $\log N$ bits long, for **SUM** word is assumed to be $\log N + \log R$ bits long, for l_p norm sketches we assume that sketches can fit in a word, for clustering a word is assumed large enough to be able to hold a single point from the metric space, and so on. Similarly, we assume it is possible to do a single word operation in one unit of time while measuring time requirements.

⁴An element is termed rare if it occurs only once (or a small number of times) in the sliding window.

Table 1 Summary of results for the sliding-window model

Problem	Space requirement (in words)	Space lower bound (in words when available)	Amortized update time	Worst case update time
BASIC COUNTING	$O(\frac{1}{\epsilon} \log N)$	$\Omega(\frac{1}{\epsilon} \log N)$	$O(1)$	$\Omega(\frac{1}{\epsilon} \log N)$
SUM	$O(\frac{1}{\epsilon} \log N)$	$\Omega(\frac{1}{\epsilon} \log N)$	$O(1)$	$\Omega(\frac{1}{\epsilon} (\log N + \log R))$
Variance	$O(\frac{1}{\epsilon^2} (\log N + \log R))$	$\Omega(\frac{1}{\epsilon} \log N)$	$O(1)$	$O(\frac{1}{\epsilon^2} (\log N + \log R))$
l_p norm sketches	$O(\log N)$	$O(\log N)$	$O(1)$	$O(\log N)$
k -median clustering ($2^{O(1/r)}$ -approximation)	$O(\frac{k}{r^4} N^{2\tau} \log^2 N)$	$\Omega(\frac{1}{\epsilon} \log N)$	$\tilde{O}(k)$	$\tilde{O}(\frac{k^2}{r^3} N^{2\tau})$
Min/Max	$O(N)$	$\Omega(N)$	$O(\log N)$	$O(\log N)$
Similarity	$O(\log N)$		$O(\log \log N)$ (w.h.p.)	$O(\log \log N)$ (w.h.p.)
Rarity	$O(\log N)$		$O(\log \log N)$ (w.h.p.)	$O(\log \log N)$ (w.h.p.)
Approximate counts	$O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon})$	$\Omega(1/\epsilon)$	$O(\log(1/\epsilon))$	$O(1/\epsilon)$
Quantiles	$O(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log N)$	$\Omega((1/\epsilon) \log(\epsilon N / \log(1/\epsilon)))$	$O(\log(1/\epsilon) \log(\epsilon N / \log(1/\epsilon)))$	$O(1/\epsilon \log(1/\epsilon))$

References

1. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in *Proc. of the 1996 Annual ACM Symp. on Theory of Computing* (1996), pp. 20–29
2. A. Arasu, G. Manku, Approximate counts and quantiles over sliding windows. Technical report, Stanford University, Stanford, California (2004)
3. B. Babcock, M. Datar, R. Motwani, Sampling from a moving window over streaming data, in *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms* (2002), pp. 633–634
4. B. Babcock, M. Datar, R. Motwani, L. O’Callaghan, Maintaining variance and k-medians over data stream windows, in *Proc. of the 2003 ACM Symp. on Principles of Database Systems* (2003), pp. 234–243
5. E. Cohen, M. Strauss, Maintaining time-decaying stream aggregates, in *Proc. of the 2003 ACM Symp. on Principles of Database Systems* (2003), pp. 223–233
6. A. Das, J. Gehrke, M. Riedwald, Approximate join processing over data streams, in *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data* (2003), pp. 40–51
7. M. Datar, Algorithms for data stream systems. PhD thesis, Stanford University, Stanford, CA, USA (2003)
8. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows. *SIAM J. Comput.* **31**(6), 1794–1813 (2002)
9. M. Datar, S. Muthukrishnan, Estimating rarity and similarity over data stream windows, in *Proc. of the 2002 Annual European Symp. on Algorithms* (2002), pp. 323–334
10. J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, An approximate l_1 -difference algorithm for massive data streams, in *Proc. of the 1999 Annual IEEE Symp. on Foundations of Computer Science* (1999), pp. 501–511
11. A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M. Strauss, Fast, small-space algorithms for approximate histogram maintenance, in *Proc. of the 2002 Annual ACM Symp. on Theory of Computing* (2002)
12. A. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *Proc. of the 2001 Intl. Conf. on Very Large Data Bases* (2001), pp. 79–88
13. M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data* (2001), pp. 58–66
14. S. Guha, N. Mishra, R. Motwani, L. O’Callaghan, Clustering data streams, in *Proc. of the 2000 Annual IEEE Symp. on Foundations of Computer Science* (2000), pp. 359–366
15. P. Indyk, Stable distributions, pseudorandom generators, embeddings and data stream computation, in *Proc. of the 2000 Annual IEEE Symp. on Foundations of Computer Science* (2000), pp. 189–197
16. J. Kang, J.F. Naughton, S. Viglas, Evaluating window joins over unbounded streams, in *Proc. of the 2003 Intl. Conf. on Data Engineering* (2003)
17. X. Lin, H. Lu, J. Xu, J.X. Yu, Continuously maintaining quantile summaries of the most recent n elements over a data stream, in *Proc. of the 2004 Intl. Conf. on Data Engineering* (2004)
18. R. Motwani, P. Raghavan, *Randomized Algorithms* (Cambridge University Press, Cambridge, 1995)
19. J.S. Vitter, Random sampling with a reservoir. *ACM Trans. Math. Softw.* **11**(1), 37–57 (1985)

Part II

Mining Data Streams

Clustering Data Streams

Sudipto Guha and Nina Mishra

1 Introduction

Clustering is a useful and ubiquitous tool in data analysis. Broadly speaking, clustering is the problem of grouping a data set into several groups such that, under some definition of “similarity,” similar items are in the same group and dissimilar items are in different groups. In this chapter, we focus on clustering in a streaming scenario where a small number of data items are presented at a time and we cannot store all the data points. Thus, our algorithms are restricted to a single pass. The space restriction is typically sublinear, $o(n)$, where the number of input points is n .

An important aspect of clustering is the definition of what constitutes a “cluster” and what makes one clustering better than another. Certainly, we seek the best possible clustering of the data, but what defines the “best”? Typically, given a clustering of the data we compute a function that maps the clustering into a number which denotes the quality of the clustering, referred to as the objective function. And thus the clustering problem becomes an *optimization* problem: *find a clustering of the best quality*. This mapping of the clustering to an objective function is an important issue—from the perspective of clustering in a streaming scenario where we have incomplete information about the evolving dataset, we have to impose a restriction that the problem remains meaningful. For example, if the cluster quality depends on all the points that are assigned to a cluster then unless there is some way of

S. Guha (✉)

Department of Computer Information Sciences, University of Pennsylvania, Philadelphia,
PA 19104, USA

e-mail: sudipto@cis.upenn.edu

N. Mishra

Computer Science Department, Stanford University, Stanford, CA 94305, USA

e-mail: nmishra@gmail.com

representing that information in small space no stream clustering is feasible. More specifically, we must be guaranteed that the clustering of the initial part of the stream is consistent with the latter part. Towards this end, we assume that we are interested in clustering data that arises from (semi) metric spaces, that is, there is a “distance” function (defined formally later) which encodes the relationship between pairs of points. The core property of “distance” is that once we compute the relationship between two points, no subsequent information affects that relationship—although subsequent information can determine if two points are to be in the same or different clusters. We will assume that the objects to be clustered are presented one at a time and in their entirety. We also assume the existence of an “oracle” or a “black-box” distance function. The oracle distance allows us to simply store these points and compute the distances between a pair of them on demand. There is no unique “correct” distance function and which function is the best suited in any given scenario is debated vigorously. Since the thrust of this chapter is the investigation of general techniques for clustering, we will not get into the issue of which distance function is better. Another advantage of assuming a black-box distance function is that the techniques will be relevant to a wide spectrum of data, with categorical, numerical, string or binary attributes.

Due to the algorithm’s constraint on storage space, we also assume that the number of clusters is bounded by some given quantity k . There is considerable discussion regarding whether it is reasonable to assume the parameter k . Some clustering objectives avoid the assumption by trading off inter-cluster distance with intra-cluster distance. Others introduce a penalty term into the function such that the quality of the clustering grows worse as the number of clusters increase. For many applications, it is desirable to represent each cluster by a single object (which is commonly referred to as a center, centroid or medoid). The quality of a single cluster is some function of the similarity/dissimilarity of the objects to the corresponding center. Center-based models are natural in a streaming scenario where it is imperative that we store an implicit representation of each cluster since we cannot store all of the objects in a cluster. In this case it is also natural that every object is implicitly assigned to the nearest representative, thereby defining a partition. The list of representatives specifies the clustering completely and is therefore a small-space description of the clustering. For the streaming algorithms described in this chapter, only this implicit representation of the clustering is produced. That is, the centers are output (and a complete partitioning of the stream is not output, but could be computed from the centers).

Approximation Guarantees

In this chapter, we will focus on algorithms that have guaranteed performance bounds. The specific avenue we pursue is *approximation algorithms* where we devise an algorithm with a proof that the algorithm performs provably close to the optimum. A formal definition can be found in the next section. The fundamental aspect of this approach is that approximation algorithms provide a guarantee, or a

certificate, of the quality of a clustering. Contrast this with a heuristic that works “well” but can have arbitrarily worse performance. In this chapter, we seek algorithms with a provable guarantee.

The goal of approximation is not only to understand the underlying mathematical structure of the optimization, but also to provide a well-grounded starting point. Every domain will have its own skew and it goes without saying that we should target our heuristics towards exploiting the properties of the domain, but building these heuristics on top of a basic structure given by an approximation algorithm provides a solid foundation. Further, a 3-approximation guarantee implies that the solution is no worse than 3 times the optimum—in practical situations, solutions may be near indistinguishable from the optimal.

Overview and Organization

The challenge of the streaming scenario is to compute in small space and in a single pass, if possible. Clustering is already a non-trivial problem without these restrictions and their presence narrows down the choices in designing algorithms. It is interesting to note that clustering itself is a way of summarizing data and the algorithms we will see in this chapter use different approaches to maintain the summary.

We begin by reviewing some basic definitions and preliminaries. We discuss the k -center problem in Sect. 3 and the k -median problem in Sect. 4. The two chosen algorithms demonstrate different techniques used in the context of streaming algorithms. The first demonstrates a clustering algorithm similar to quantile estimation on data streams where at all points we maintain a “proof” that the centers we compute are near optimum. In this sense the algorithm is an online algorithm. In the case of the k -median problem, we focus on a divide-and-conquer approach to streaming algorithms. Another tool that is very effective at clustering streams is to simply maintain a small random sample of the stream that is clustered whenever a clustering of the entire stream is desired. Sampling is discussed in Sect. 5. Also, many references to relevant papers are described in that section.

2 Preliminaries and Definitions

To define the types of data sets we consider in this chapter, we introduce the notion of a (semi-)metric space.

Definition 1 For a set of points X and a distance measure D , we say that (X, D) is a *semi-metric* if the following criteria are satisfied:

1. $D(x, x) = 0$ for all $x \in X$;
2. (Symmetry) $D(x, y) = D(y, x)$ for all $x, y \in X$;
3. (Triangle Inequality) $D(x, z) \leq D(x, y) + D(y, z)$ for all $x, y, z \in X$.

If we also have $[D(x, y) = 0] \Rightarrow [x = y]$ then (X, D) defines a metric space.

In reality, a semi-metric space can be viewed as a metric space with possible duplicates. One common example of a metric space is the Euclidean distance for points in \mathbb{R}^d . We denote this metric space by (X, L_2) where $L_2(x, y) = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}$ for $X \subset \mathbb{R}^d$. All data sets considered in this chapter are assumed to be metric spaces.

We describe two clustering objectives functions. These objectives serve as good illustrations for how streams can be clustered. This chapter does not provide an exhaustive list of objectives.

If we view the clusters as “balls”, the quality of the clustering can be thought of as the largest radius of the “ball” required to cover all the objects of a cluster. As a natural extension, the overall clustering objective is to choose k centers and find the minimum radius r such that every point will belong to a “ball” of radius at most r around these objects. If the centers are restricted to be points in the input data set the clustering problem is known as the k -center problem. More formally,

Definition 2 (The k -Center Problem) Given an integer k and a set of points S from a metric space (X, D) where $|S| = n$, the k -center problem seeks to find k representative points $C = \{c_1, \dots, c_k\} \subseteq S$ such that

$$\max_{x \in S} \min_{c_i \in C} D(c_i, x)$$

is minimized.

Observe that if we specify the centers or representatives then each point should be assigned to the cluster corresponding to the nearest center (the inner minimization achieves this). The outer maximization identifies the largest radius required. In a geometric setting, the definition can be relaxed to include any point in the space as a possible center.

The k -center objective depends on only one number, the farthest distance from any point to its nearest center, alternatively the radius of the “fattest” cluster. The objective ignores the distance from all other points to their nearest center. Consequently, if one outlier point is very far from the other points, that outlier will determine the quality of the clustering (so that it does not really matter how the other points are clustered).

An alternate objective, that is less sensitive to outliers, is to minimize the sum of distances from points to nearest centers, known as the k -median problem. The objective seeks to minimize the “average” radius of a ball required to cover all of the points.

Definition 3 (The k -Median Problem) Given an integer k and a set S of n points from a metric space (X, D) , the k -median problem seeks to find k representative points $C = \{c_1, \dots, c_k\}$ such that we minimize

$$\min_{x \in S} \sum_{c_i \in C} \min D(c_i, x).$$

The k -median objective is closely related to the *squared error distortion* or *k -median-squared* objective

$$\min \sum_{x \in S} \min_{c_i \in C} D(c_i, x)^2$$

where the centers c_i are typically the mean of all the points in the cluster i . In the case that the objects to be clustered are points in \mathbb{R}^d , the k -Means algorithm is often used in practice to identify a local optimum solution.

As mentioned earlier, for most natural clustering objective functions, the optimization problems turn out to be NP-hard. Therefore, we seek to design algorithms that guarantee a solution whose objective function value is within a fixed factor of the value of the optimal solution. We use the following definition of approximation algorithms.

Definition 4 A ρ -approximation algorithm is defined for a minimization problem to be an algorithm that gives an approximate solution which is at most ρ times the true minimum solution. ρ is referred to as the approximation ratio or approximation factor of the algorithm.

In fact, finding a 1.3-approximation algorithm for k -median is also NP-hard. The provable approximation ratios for k -median are small constants. In the context of streaming, given the one-pass, small-space restrictions, the approximation ratio tends to be larger than the best known polynomial-time algorithms that do not have such restrictions.

3 The k -Center Clustering on a Stream

In order to appreciate the difficulty that arises in clustering a data stream, we describe the non-streaming algorithms first. We begin with the FARTHEST POINT clustering algorithm which is given in Fig. 1. The algorithm produces a collection of k centers and a 2-approximation to the optimal clustering, i.e., finds a set of k centers such that the maximum distance from a point to its nearest center is at most twice as large as the optimal k -center clustering. The algorithm finds the collection via an iterative process where the first center is chosen arbitrarily and subsequent centers are selected by finding the point in the data set farthest from its closest center. Recall again that a point here refers to an object in an arbitrary metric space, and not just points in Euclidean space.

Suppose the algorithm had been run one more step. Suppose it would have chosen c_{k+1} and the distance from c_{k+1} to the other centers would have been t . Then we would have $k + 1$ points, c_1, \dots, c_k, c_{k+1} , such that the distance between any pair of them is at least t (otherwise c_{k+1} should have preceded c_k). Now by the pigeonhole principle, at least two of these $k + 1$ points must belong to the same cluster. The radius of the cluster in the optimal solution which contains two of the c_i 's is at

FARTHEST POINT ALGORITHM

1. Input: Set of points (objects) S , distance metric (X, D) , number of centers k
2. Output: Set of k centers
3. $c_1 \leftarrow$ any point in S
4. for $i = 2$ to k
 - (a) $c_i \leftarrow$ point in S farthest from c_1, \dots, c_{i-1}
5. Output c_1, \dots, c_k

Fig. 1 The FARTHEST POINT algorithm for k -center

least $\frac{t}{2}$, otherwise the two points would be distance less than t apart. All that remains is to observe that every point is within distance t from the chosen k centers, namely c_1, \dots, c_k . Thus we have a 2-approximation.

Theorem 1 *The FARTHEST POINT algorithm produces k centers that are a 2-approximation to the optimal k -center clustering.*

An interesting and useful observation that follows from the argument just given is that if the algorithm is run with $k = n$ we get a canonical reordering of the objects such that the first k points define an approximate k -center solution for every $1 \leq k \leq n$.

A 2-approximation algorithm is provably the best possible in case of the k -center problem. As we explain next, the problem of finding a $(2 - \epsilon)$ -approximation for $0 < \epsilon < 1$ is NP-hard since the Dominating Set problem reduces to this problem. Given a graph $G = (V, E)$, a subset of vertices S is a *dominating set* if each vertex v in V is either in S or adjacent to some vertex in S . The Dominating Set problem is, given a graph $G = (V, E)$ and an integer k , to determine if there exists a dominating set of size k . We can transform a graph G into a metric space by making the distance between adjacent vertices 1 and the distance between non-adjacent vertices 2. Observe that if G has a dominating set of size k , then there is a k -center clustering with cost 1 and if G does not have a dominating set of size k , then there is a k -center clustering of cost 2. Thus, if there was a $(2 - \epsilon)$ -approximation algorithm for k -center, we could use it to determine whether the graph has a dominating set of size k . Since the dominating set problem is NP-hard, so is the problem of finding a $(2 - \epsilon)$ -approximation to k -center.

Theorem 2 *The problem of finding a $(2 - \epsilon)$ -approximation to the k -center clustering problem is NP-hard for any $0 < \epsilon < 1$.*

The FARTHEST POINT algorithm is not suitable for clustering a stream since it is not possible to determine which point is farthest from one of the current centers without reading and maintaining information about the entire stream. However, the algorithm does demonstrate that the k centers discovered plus the next center that would be chosen (had the algorithm been run one more step) serve as a *witness* to a

HIERARCHICAL AGGLOMERATIVE CLUSTERING (HAC)	
1.	Input: Set of points (objects) S , distance metric (X, D) , number of centers k
2.	Output: Set of k centers
3.	Initially each point is a cluster (of size 1), and they are said to <i>represent</i> their cluster.
4.	for $i = 1$ to n
	(a) Pick the two closest clusters, e.g., the pair of representatives that are closest.
	(b) Merge the two clusters and choose an appropriate representative of the combined cluster.
5.	Output the tree of clusters/dendrogram.

Fig. 2 The structure of a typical hierarchical agglomerative algorithm

clustering—if we were to find $k + 1$ objects separated by distance at least t by some other method, we would still have an approximation algorithm for k -center.

One simple idea for a streaming k -center method is to always maintain k cluster representatives, assign each incoming point to its nearest representative, and then somehow update the cluster representative. The idea comes from one of the earliest and frequently used methods for clustering, Hierarchical Agglomerative Clustering (HAC), which we describe next.

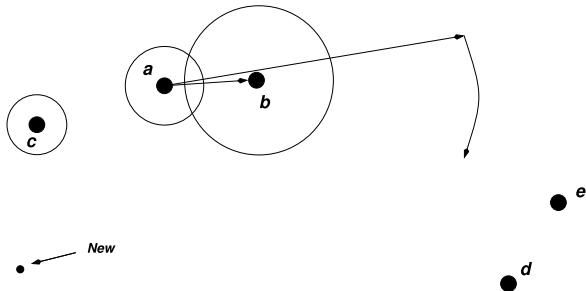
HAC is a bottom-up algorithm that repeatedly merges the two closest clusters (see Fig. 2). The algorithm produces a tree of clusters where each node is a cluster consisting of subclusters defined by its children. The leaves of the tree correspond to the individual points. The tree of clusters is often referred to as a *dendrogram*.

One naive, greedy way that HAC can be extended to handle a data stream is as follows. Always maintain k centers, and when a new point arrives, treat that point as a new center and make some greedy decision as to how best to reduce the $k + 1$ centers down to k . In particular, the two closest centers could be merged and the new center of the merged cluster is the old center with the larger radius. Regrettably, this greedy algorithm does not have a good approximation ratio—we can construct examples where the approximation ratio of the algorithm is $2k - 1$.

Consequently, we will have to be more clever than greedy if we wish to find a streaming k -center algorithm. By combining the ideas behind the FARTHEST POINT algorithm and HAC, it is possible to obtain an algorithm with good approximation ratio. The basic intuition comes from the following question: Suppose $k = 2$. If we find three representatives equidistant from each other which two do we merge? A natural answer is that all three clusters should be merged together. But what if the distances are not the same? The main intuition is to merge the closest pair of centers, but to extend the radius by a factor to account for the distances that are close. This is shown in Fig. 3 where the centers a, b are the closest, and we proceed to merge a, b, c and the new point together. Now, because we have increased the radius of the clusters significantly, it stands to reason that we should merge the centers d and e .

The overall algorithm is given in Fig. 4. The idea is to maintain a lower bound r on the optimal solution of the k -center problem, then arbitrarily select any center and merge all centers within distance r , repeating until all centers are covered. This

Fig. 3 The intuition behind the streaming k -center algorithm



DOUBLING ALGORITHM

1. Input: Sequence of points (objects) S from a metric space (X, D) arriving one at a time
2. Output: Set of k centers
3. Initially, the first k points are chosen as centers and $r \leftarrow 0$.
4. for every new point i
 - (a) Suppose the current centers are c_1, \dots, c_ℓ .
 - (b) If i is within distance $4r$ from any center, assign i to that cluster (choose arbitrarily if there are more than one).
 - (c) Otherwise,
 - i. if $\ell < k$ make $c_{\ell+1} = i$, i.e., start a new cluster with i as a center.
 - ii. Otherwise
 - A. Let the smallest distance between any pair of points in the set $C = \{c_1, \dots, c_k, i\}$ be t .
 - B. $r \leftarrow \frac{t}{2}$.
 - C. Pick a point from C and let this be a new center c'_1 . Remove all points from C that are within distance $4r$ from this center c'_1 . All the clusters (corresponding to the removed centers and possibly the singleton cluster i) are merged into a cluster with center c'_1 .
 - D. The above step is repeated until C is empty. These centers are carried over to handle the next point.
 5. Output the remaining cluster centers.

Fig. 4 Stream k -center clustering, the DOUBLING ALGORITHM

parameter r will grow monotonically as we see more points. It can be shown that the algorithm will have *at most* k centers at any point. Interestingly, the number of centers may drop below k (there may even be just 1 center), but in such a case, it can be shown that the algorithm's performance is still close to optimal.

The reduction from $k + 1$ centers to at most k centers in Step 4(c)ii balances between the greedy streaming HAC algorithm described above and the farthest point algorithm. The step is reminiscent of center greedy since it greedily decides which centers to merge. On the other hand, the lower bound on the radius given by the farthest point algorithm is cleverly used to decide which centers to merge.

It can be shown that at any point of time if the centers are $\{c_1, \dots, c_\ell\}$, where $\ell \leq k$ then the minimum pairwise distance is $4r$. Thus, when we get to the situation where we have $k+1$ points then the minimum pairwise distance, t , is greater than $4r$. Thus, when we set $r \leftarrow t/2$ the value or r at least doubles.

Now suppose the maximum radius of a cluster is $R(r)$ for a particular value of r during the run of the algorithm. When we set $r \leftarrow \frac{t}{2}$, suppose the lower bound changes from r' to r . The recurrence that describes the new maximum radius is

$$R(r) \leq R(r') + 4r.$$

The equation follows from the fact that the center of one of the merged clusters (say with center u) is at a distance $4r$ from the new center. Any point in that cluster of u can be at most $R(r')$ away from u . Thus the maximum radius of the new cluster follows from the triangle inequality. The only guarantee we have is that $r' \leq r/2$ and thus any function $R()$ satisfying

$$R(r) \leq R(r/2) + 4r$$

will upper bound the maximum radius of a cluster. The solution to the recurrence is $R(r) \leq 8r$. By inspection we get the following:

Theorem 3 *The STREAM k -CENTER CLUSTERING algorithm given in Fig. 4 finds an 8-approximation to the k -center streaming problem using $O(k)$ space and $O(\tau nk)$ running time, where the cost of evaluating the distance between two objects is τ .*

One way of viewing the above algorithm is that when we decide to merge clusters, we increase the radius to be a bit more than the minimum required to reduce the number of centers to k . In this process, we may have less than k centers. The free centers are used to start new clusters if a new object is very far from the current set of objects. In a sense, the algorithm provisions for the future, and incurs a cost in the current solution. This is a very useful paradigm in streaming algorithms.

4 The k -Median Divide-and-Conquer Approach

Recall that the k -median objective is, given a set of objects S , to find a set C of medians (centers) such that $C \subseteq S$, $|C| \leq k$ and $\sum_{x \in S} \min_{c \in C} D(x, c)$ is minimized. This objective is sometimes referred to as the discrete k -median problem since the centers are restricted to be points in the data set. In the event that the objects come from a continuous space, say (\mathbb{R}^d, L_2) , it is natural to ask whether it is possible to identify centers $C \subset \mathbb{R}^d$ such that the average distance from a point to its nearest center is minimized. This problem is known as the continuous k -median problem. As it turns out, the best discrete solution is at most twice as large as the best continuous solution. Consequently, we henceforth address the discrete version of the clustering problem.

1-SWAP ALGORITHM

1. Input: Set of points S , distance metric d , number of centers k
2. Output: Set of k centers
3. Let C be a set of k centers from S .
4. While there exists a point $x \in S$ and $y \in C \setminus S$ such that $\sum_{x \in S} \min_{c \in C \setminus \{x\} \cup \{y\}} D(x, c)$ is smaller than $\sum_{x \in S} \min_{c \in C} D(x, c)$ by a factor of $(1 + \epsilon)$
 - (a) $C \leftarrow C \setminus \{x\} \cup \{y\}$
5. Output C

Fig. 5 The local search based SWAP algorithm

Theorem 4 *For any collection C of k centers that are a continuous k -median solution of cost T , there exists a discrete k -median solution of cost $2T$.*

Proof Imagine a particular continuous cluster consisting of points x_1, \dots, x_p with continuous center y . If we replace the continuous center y with the closest discrete center, say x_i , then the distance from any point to its nearest center at most doubles since $D(x_j, x_i) \leq (D(x_j, y) + D(y, x_i)) \leq 2D(x_j, y)$. Thus, summing over all points (and all clusters), the total cost at most doubles. \square

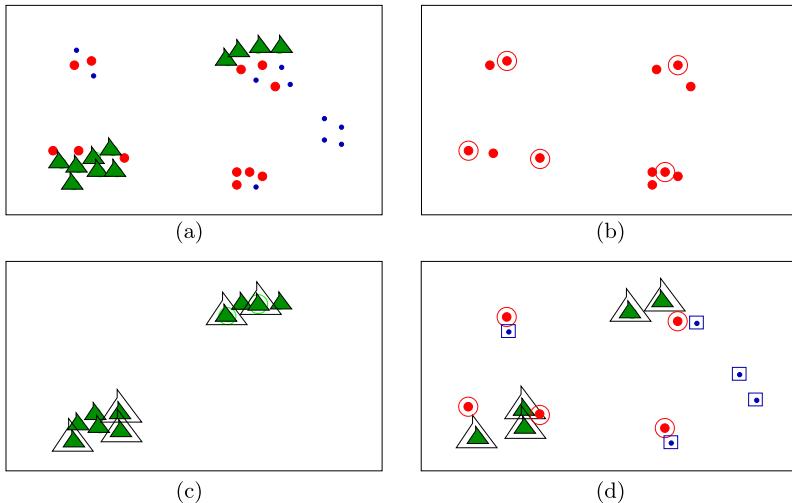
Also closely related to the discrete k -median objective is the k -median-squared objective where the goal is to minimize the average squared distance from a point to its nearest center. We delve more into this objective in Sect. 4.2. As before, we begin by describing a non-streaming solution to the k -median problem. The algorithm starts with an initial set of k centers and then repeatedly swaps one of the current centers with a non-center if the clustering quality improves (see Fig. 5).

There are two important issues with respect to the Local Swap Algorithm, running time and clustering quality. If the initial set of centers are chosen via the FARTHEST POINT algorithm, then it can be shown that the algorithm's running time is polynomial in n and k . Note that we did not specify how the initial centers are chosen in Step 3 of the Local Swap algorithm because any method may be used. However, to ensure that the algorithm runs in polynomial time, one needs to start with a solution that is not arbitrarily far from the optimum. If the FARTHEST POINT algorithm is used as a starting point, then one can show that such a set of initial centers is no more than a factor of n away from the optimal solution. This fact is helpful in bounding the algorithm's running time. In terms of final clustering quality, it can be shown that the algorithm finds a 5-approximation (ignoring ϵ -factors). If the swap in Step 4 is performed with p points at a time then the algorithm can be shown to be a $(3 + 2/p)$ -approximation (again, ignoring ϵ -factors). The algorithm does not lend itself to streaming data—multiple passes are essential to evaluate the cost of the solution requiring that all points be stored. But maintaining all points in main memory is not feasible in a streaming scenario.

We next describe a simple divide-and-conquer approach that yields a good k -median clustering of a data stream. The basic idea is to break the stream into pieces and cluster each piece. For each piece, the cluster centers and a count of the

***k*-MEDIAN STREAM ALGORITHM**

1. Input: Data Stream $S = x_1, \dots, x_n$, distance metric d , number of centers k
2. Partition the stream into consecutive pieces P_1, \dots, P_m
3. For each piece P_i
 - (a) Cluster P_i
 - (b) In memory, maintain centers of P_i and the number of points (weight) assigned to (nearest to) each center.
4. To output a clustering after any piece P_i , cluster the weighted centers of P_1, \dots, P_i .

Fig. 6 The STREAM algorithm**Fig. 7** An illustration of the algorithm STREAM, $m = 3$ pieces and $k = 5$ clusters

number of points assigned to each center is the only information retained. In other words, the actual objects of the data stream that belong to the piece can be permanently discarded. If at any point we need to produce a final clustering, we consider the union of the (weighted) cluster centers collected from all the pieces seen so far. We cluster this set of weighted points to obtain a good clustering.

Observe that the k -Median Stream Algorithm given in Fig. 6 is really just a general framework in that the clustering in Steps 3(a) and 4 can be performed using any clustering algorithm. Note that it is possible that the weighted centers maintained in memory themselves exceed the size of memory. In such a case, we can cluster the weighted centers and use those k weighted centers as a snapshot of the stream. We illustrate the process in Fig. 7. For the sake of simplicity, the stream is partitioned into three pieces.

In Fig. 7, the entire dataset is partitioned into three pieces. The shape of a point (dot, circle or triangle) indicates the piece that the point belongs to—shown in Fig. 7(a). Figure 7(b) shows the 5-clustering of the first piece and Fig. 7(c) of the

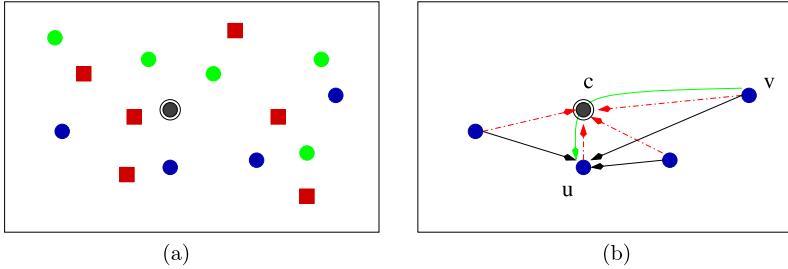


Fig. 8 A single cluster and a particular piece

second piece. (The third piece alone is not shown.) The union of all the cluster centers from the different pieces are shown in Fig. 7(d)—although the figure does not show the weights, the union of centers mimic the distribution of points in the original dataset quite well.

The importance of the natural divide and conquer algorithm is twofold. First, we can bound the quality of the clustering achieved. Second, the space required by the above algorithm is sublinear, i.e., $o(n)$. The latter can be derived as follows. Suppose we partition the stream into m equal pieces. While clustering P_i we only need to store the piece P_i and the set of centers from the previous pieces. Thus the space required is $O(mk + \frac{n}{m})$. Setting $m = \sqrt{n/k}$ the space requirement is $O(\sqrt{nk})$ which is $o(n)$ since typically $k \ll n$. The benefit carries over to the running time as well. Most clustering algorithms with approximation guarantees require at least quadratic time. The running time of the streaming algorithm, using such a quadratic-time clustering subroutine, is $O(m \frac{n^2}{m^2})$ which is $O(n^{1.5} k^{0.5})$.

The quality of the clustering algorithm is given in the next theorem. The constants in the theorem can be improved (see the Notes section at the end of the chapter). We prove a weaker result that brings out the key ideas.

Theorem 5 *If the stream is a sequence of points S and if an optimal algorithm is used to identify the cluster centers in Steps 3(a) and 4 of the k -Median Stream Algorithm, then for any i , the cluster centers output for P_1, \dots, P_i are an 8-approximation to the optimal clustering.*

Moreover, if we use a α -approximation for the clustering substeps then the final solution is a $2\alpha(2\alpha + 1) + 2\alpha$ -approximation algorithm.

Proof Sketch To see how the result is derived, consider a single cluster as in Fig. 8(a). Consider all the objects in a particular piece P_i and the center of the cluster as in Fig. 8(b), note that the center of the cluster c may not belong to this piece. But if we were to choose the point u , which is closest to c among all points of the piece, then the distance $D(u, v)$ for any other point v can be bound by the sum $D(u, c) + D(c, v)$ which is at most $2D(c, v)$. Thus adding over all v , over all clusters (equivalent to adding over all points in P_i), we can say that there is a clustering that costs at most twice the cost of the particular piece P_i in the optimal

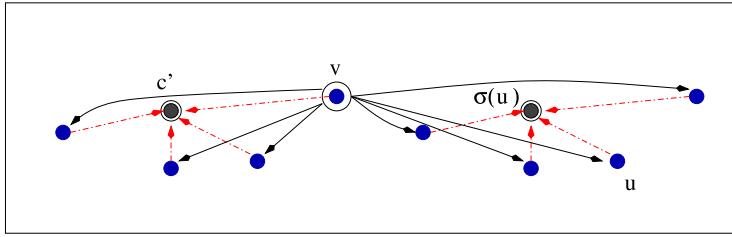


Fig. 9 The overall analysis

solution. This technique is known as a shifting argument—we show that there exists a good solution if we consider some hypothetical point, and “shift” to the actual point closest to the hypothetical one.

But we cannot control the clustering of the piece P_i , and the actual clustering may merge parts of two clusters which contains points from P_i . What we can guarantee is that the overall cost of the clustering substep, over all pieces, is at most $2OPT$. If we use an α -approximation algorithm the cost will be $2\alpha OPT$.

Now consider putting together the cluster centers from the pieces. Suppose our clustering chose v as one of the centers. We will show that there exist hypothetical k centers such that the cost of clustering the weighted centers is bounded by βOPT . Thus, by the *shifting argument*, we guarantee a cost of $2\beta OPT$.

Consider the center $\sigma(v)$ in the optimal solution closest to v , this object may not have been chosen as a center while clustering the pieces. Thus, $\sigma(v)$ may not exist as a weighted center and in that sense is “hypothetical”. Let $\sigma(u)$ denote the center to which u is assigned in the optimal solution. The situation is represented in Fig. 9.

Now the distance $D(v, \sigma(v))$ can be bounded above by the sum of $D(v, u) + D(u, \sigma(u))$:

$$D(v, \sigma(v)) \leq D(v, \sigma(u)) \leq D(v, u) + D(u, \sigma(u)).$$

Assume that the number of objects in the cluster of v is w_v . If we sum the above equation over all u in the cluster of v we get:

$$w_v D(v, \sigma(v)) \leq \sum_{u \in \text{Cluster}(v)} D(v, u) + D(u, \sigma(u)).$$

Now if we were to sum over all v , then

$$\sum_v w_v D(v, \sigma(v)) \leq \sum_v \sum_{u \in \text{Cluster}(v)} D(v, u) + \sum_u D(u, \sigma(u)).$$

The left hand side is the cost of clustering the weighted centers. The last term in the right hand side is the cost of the optimal solution! The first term is the cost of clustering the pieces, which is at most $2\alpha OPT$. Therefore, there exists a hypothetical cost of $(2\alpha + 1)OPT$. This implies that there exists a solution of cost $2(2\alpha + 1)OPT$. But we cannot guarantee that we achieve that solution and we suffer a further α factor due to the approximation.

The total cost is $2\alpha OPT$, which we paid to cluster the pieces, plus the cost of the weighted centers, which is $(2\alpha + 1)\alpha OPT$. One way to view it is that if the point u was looking for a center to get assigned to, we send u to v accruing a cost of $2\alpha OPT$ and then from v to the final center accruing a cost of clustering the weighted centers. \square

By a recursive application of the above theorem it also follows that with each additional level of clustering, i.e., the pieces themselves are split using the two level algorithm, the algorithm will give up an extra multiplicative factor for each step of the process. The constants can be improved significantly by using a different local search algorithm, see [9] in Step 3(a) of the STREAM algorithm. The recursive algorithm gives a $O(nk \log n)$ time algorithm that uses space $O(n^\epsilon)$ and gives a $2^{O(\frac{1}{\epsilon})}$ -approximation algorithm in a single pass over the data. A full proof of the above requires development of ideas from facility location problems, and thus is omitted. Note that the space requirement can be further improved to $(k \log n)^{O(1)}$ while still achieving a constant factor approximation, see the Notes section.

4.1 Performance of the Divide-and-Conquer Approach

The above algorithm is a natural algorithm, and similar ideas of partially clustering data have been used in practice. In one such algorithm, BIRCH [38], a cluster feature tree (CFTree) preclusters the data. The tree is a space partitioning tree and every input point follows a path along the tree based on the membership of the point to the two sides of the partition corresponding to a node. The tree is adjusted dynamically and balanced so that roughly the same number of points are allocated to the leaves. The leaf nodes maintain a summary and these summaries are the preclusters. BIRCH then uses a standard clustering algorithm, HAC, k -Means (see Sect. 4.2), etc., to complete the clustering. Most known preclustering algorithms are a similar top down algorithm, i.e., points are classified into bins based on a decision tree.

The divide-and-conquer algorithm yields a *bottom up* preclustering. From the point of view of the amount of information used by the algorithm, a bottom up clustering has the advantage that the decision to group points in a cluster is taken after a larger number of points are seen in the stream. It is quite natural to expect that the above bottom up process yields a more robust (pre)clustering at the expense of taking more time, compared to a top down process.

4.2 The k -Median-Squared Objective and the k -Means Algorithm

The k -median-squared objective function seeks to minimize the function $\sum_{x \in X} \min_{c \in C} D^2(x, c)$ where $|C| = k$. This objective has a rich history dating back to the beginning of Location Theory put forth by Weber in 1909 [37]. Many

known constant factor approximation algorithms for the k -median problem yield a constant factor approximation for the sum of squares objective as well using a “relaxed triangle inequality”. This relaxed triangle inequality states that for all x, y, z , $D^2(x, z) \leq 2(D^2(x, y) + D^2(y, z))$. The reason this inequality holds is that $D^2(x, z) \leq (D(x, y) + D(y, z))^2 \leq 2(D^2(x, y) + D^2(y, z))$. Note that there is no known generic method for converting an approximation bound for k -median to an approximation bound for k -median-squared. Rather, the relaxed triangle inequality can sometimes be invoked in steps of the proof where the triangle inequality is used.

For historical reasons and because the algorithm is widely used, we describe the **k -Means** algorithm or *Lloyds algorithm*, which attempts to solve the continuous version of the k -median-squared objective. The algorithm starts with k tentative centers. Each of the data objects are assigned to their nearest center. After all points are assigned, the mean of all the points assigned to a center forms the new center of the cluster. These k centers now serve as the new set of tentative centers. The process repeats until the sum of squares does not go down any further (or much further). The final k centers define the clustering.

The above is an iterative improvement algorithm, but is not guaranteed to converge on arbitrary data. In fact, for poor placement of initial centers, there exists situations where the algorithm provably finds solutions that are arbitrarily worse than the optimum. For example, suppose that there are four two-dimensional points $(-M, -1), (-M, 1), (+M, -1), (+M, 1)$ to be 2-clustered and the initial centers are $(0, 1)$ and $(0, -1)$. In this case, the k -Means algorithm will output $(0, 1)$ and $(0, -1)$ as the final centers. The cost of this solution is $4M^2$ as opposed to the optimal 4. Thus by pushing the points farther apart, i.e., increasing M , the ratio of the k -means solution to the optimal solution is arbitrarily bad.

However, if the k -Means algorithm is properly seeded with a good collection of initial centers, then the clustering quality is not arbitrarily worse than the optimal. For a specific randomized method of selecting initial centers [4], it can be shown that the expected cost of the initial centers (without running k -Means) is a $O(\log k)$ -approximation. If k -Means is subsequently run with this choice of initial centers, then the final clustering cost only improves.

5 Notes

The first approximation algorithm for the k -center clustering problem was provided in [23]; the farthest point algorithm is presented in [18]. The problem is hard to approximate up to a $(2 - \epsilon)$ -factor for general metric spaces, and this does not improve significantly in case of geometric spaces. In [17], it is shown that k -center is hard to approximate up to a factor of 1.82 in the plane.

The k -center stream clustering algorithm given in Sect. 3 is presented in [8]. While that algorithm gives an 8-approximation, a randomized version yields an improved $2e$ -approximation. The authors also show that greedy algorithms for maintaining a collection of k centers perform poorly. Finally, lower bounds demonstrate

that no deterministic stream k -center clustering algorithm can achieve a bound of better than $(1 + \sqrt{2})$ for arbitrary metric spaces.

For the k -median problem several approximation algorithms were known [10, 25, 32]—the analysis of 1-swap (or p -swap) is due to [5]. The swap based algorithms were introduced in [27] as ‘Partition Around Medoids’ (PAM) in the context of the k -median-squared objective. Bicriterion approximation algorithms, i.e., algorithms using slightly more than k medians were provided in [9, 29, 30]. Several of the above algorithms extend naturally to the sum of squared distances objective function. Approximation algorithms specifically tuned for the squared error objective were studied in [26] for arbitrary metric spaces. The running time of the approximation algorithm has been an issue—see the discussion on sampling based approaches below.

In the Euclidean case, [28] gave a near linear time approximation scheme for the k -median problem, based on a previous result by [3]. Using core-sets introduced in [1], the running time has been further reduced to linear in n (plus a $(\frac{k \log n}{\epsilon})^{O(1)}$ term and a function dependent on k, ϵ, d) in [21]. The algorithm of [21] extends to a similar result for the sum of squared objective as well. The authors of [15] demonstrate that the Singular Value Decomposition of the $n \times d$ matrix representing n points in \mathbb{R}^d can be used to find a 2-approximation to the optimum for the sum of squared objective.

The algorithm given in Sect. 4 is a slight variation of the first k -median streaming algorithm presented in [19]. Their paper gives a 5-approximation for a simple 2-level algorithm, and in general, describes a one-pass $2^{O(1/\epsilon)}$ -approximation algorithm that uses $O(n^\epsilon)$ space. Subsequently, in [12] a constant factor, one-pass, randomized stream algorithm using $(k \log n)^{O(1)}$ space is provided. In the sliding window model a $2^{O(1/\tau)}$ -approximate k -median solution for the last N points using $O(\frac{k}{\tau^4} N^{2\tau} \log^2 N)$ space, where the parameter $\tau < 1/2$ enables a tradeoff between space and clustering quality, is given.

While we only discussed single-pass algorithms, there is certainly strong motivation for multi-pass algorithms. For instance, if data is stored in some slow but large repository, a multi-pass algorithm may be desirable. In such a scenario, it may be important to trade off resources required for preprocessing the data with resources required for clustering the data. The boundary between “streaming” and “offline” computation is then somewhat blurred.

The Sampling Approach

Given that random sampling is a powerful tool for stream processing and given that a random sample can be maintained using Vitter’s reservoir sampling technique (refer to Chap. 1), a natural question is can one effectively cluster a stream by simply maintaining a random sample? The general answer is that a small random sample can be shown to be descriptive of all the data.

For the k -center objective, if clustering is based on a random sample, then observe that it is not possible to obtain a constant-factor approximation of the entire

stream since there may be one point in the stream that is very far from every other point in the stream. The probability that such a faraway point is in a small random sample is very small. However, if the algorithm is allowed to ignore a small fraction of the points, then constant factor approximations can be obtained via random sampling [2] in a single pass. Clustering algorithms that discount a small fraction of points are considered in [11] in greater detail.

For the k -median objective, in [24] a bicriteria approximation, which identifies $2k$ centers that are a constant factor approximation to the best k -median clustering, was provided. The two-pass algorithm requires two samples of size $\tilde{O}(\sqrt{nk})$. In [34], it is shown that the simple single-pass algorithm of drawing a random sample and then running an approximation algorithm on the sample yields a good approximation to the best clustering of all the data. The sample size was shown to depend polynomially on the ratio of the diameter of the data to the desired error and depend logarithmically on n . Further sampling results can be found in [6, 14, 33], and the references therein. Also, fast approximation algorithms for the k -median problem have been devised through recursive sampling [31, 35].

As mentioned earlier, all the above assume that the distances form a semi-metric, i.e., satisfy the triangle inequality and symmetry. Without the triangle inequality assumption, it is NP-hard to find any constant factor approximation to most clustering objectives including the k -center and k -median objectives [22, 36]. The same holds for the k -center problem if the distances are asymmetric (but satisfy triangle inequality) [13].

There are also many heuristics for clustering streams. BIRCH [38] compresses a large data set into a smaller one via a CFTree (clustering feature tree) as we discussed earlier. CURE [20] uses sampling and multi-centroid representations to cluster the data. Both of the above algorithms output the cluster centers in one pass. Several papers, e.g., [7, 16], propose to repeatedly take k weighted centers (initially chosen randomly with weight 1) and as much data as can fit in main memory, and compute a k -clustering. The new k centers so obtained are then weighted by the number of points assigned, the data in memory is discarded and the process repeats on the remaining data. This algorithm places higher significance on points later in the data stream.

References

1. P.K. Agarwal, S. Har-Peled, K.R. Varadarajan, Approximating extent measure of points. J. ACM **51**(4), 606–635 (2004)
2. N. Alon, S. Dar, M. Parnas, D. Ron, Testing of clustering. SIAM J. Discrete Math. **16**(3), 393–417 (2003)
3. S. Arora, P. Raghavan, S. Rao, Approximation schemes for Euclidean k -medians and related problems, in *Proc. of STOC* (1998), pp. 106–113
4. D. Arthur, S. Vassilvitskii, k -Means++: the advantages of careful seeding, in *Proc. of SODA* (2007)
5. V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, V. Pandit, Local search heuristic for k -median and facility location problems. SIAM J. Comput. **33**(3), 544–562 (2004)

6. S. Ben-David, A framework for statistical clustering with a constant time approximation algorithms for k -median clustering, in *Proc. of COLT* (2004), pp. 415–426
7. P.S. Bradley, U.M. Fayyad, C. Reina, Scaling clustering algorithms to large databases, in *Proc. of KDD* (1998), pp. 9–15
8. M. Charikar, C. Chekuri, T. Feder, R. Motwani, Incremental clustering and dynamic information retrieval. *SIAM J. Comput.*, 1417–1440 (2004)
9. M. Charikar, S. Guha, Improved combinatorial algorithms for the facility location and k -median problems, in *Proc. of FOCS* (1999), pp. 378–388
10. M. Charikar, S. Guha, É. Tardos, D.B. Shmoys, A constant factor approximation algorithm for the k -median problem. *J. Comput. Syst. Sci.* **65**(1), 129–149 (2002)
11. M. Charikar, S. Khuller, D.M. Mount, G. Narasimhan, Algorithms for facility location problems with outliers, in *Proc. of SODA* (2001), pp. 642–651
12. M. Charikar, L. O’Callaghan, R. Panigrahy, Better streaming algorithms for clustering problems, in *Proc. of STOC* (2003), pp. 30–39
13. J. Chuzhoy, S. Guha, E. Halperin, S. Khanna, G. Kortsarz, R. Krauthgamer, J. Naor, Asymmetric k -center is $\mathcal{Q}(\log^* n)$ hard to approximate, in *Proc. of STOC* (2004), pp. 21–27
14. A. Czumaj, C. Sohler, Sublinear-time approximation for clustering via random sampling, in *Proc. of ICALP* (2004), pp. 396–407
15. P. Drineas, A. Frieze, R. Kannan, S. Vempala, V. Vinay, Clustering large graphs via the singular value decomposition. *Mach. Learn.* **56**, 9–33 (2004)
16. F. Farnstrom, J. Lewis, C. Elkan, True scalability for clustering algorithms, in *SIGKDD Explorations* (2000)
17. T. Feder, D.H. Greene, Optimal algorithms for appropriate clustering, in *Proc. of STOC* (1988), pp. 434–444
18. T.F. Gonzalez, Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.* **38**(2–3), 293–306 (1985)
19. S. Guha, N. Mishra, R. Motwani, L. O’Callaghan, Clustering data streams, in *Proc. of FOCS* (2000), pp. 359–366
20. S. Guha, R. Rastogi, K. Shim, CURE: an efficient clustering algorithm for large databases, in *Proc. of SIGMOD* (1998), pp. 73–84
21. S. Har-Peled, S. Mazumdar, On coresets for k -means and k -median clustering, in *Proc. of STOC* (2004), pp. 291–300
22. D.S. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems* (Brooks/Cole, Pacific Grove, 1996)
23. D.S. Hochbaum, D.B. Shmoys, A unified approach to approximate algorithms for bottleneck problems. *J. ACM* **33**(3), 533–550 (1986)
24. P. Indyk, Sublinear time algorithms for metric space problems, in *Proc. STOC* (1999)
25. K. Jain, V. Vazirani, Approximation algorithms for metric facility location and k -median problems using the primal–dual schema and Lagrangian relaxation. *J. ACM* **48**(2), 274–296 (2001)
26. T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, A.Y. Wu, A local search approximation algorithm for k -means clustering, in *Proc. of SoCG* (2002), pp. 10–18
27. L. Kaufmann, P.J. Rousseeuw, Clustering by means of medoids, in *Statistical Data Analysis Based on the L₁ Norm and Related Methods* (Elsevier Science, Amsterdam, 1987), pp. 405–416
28. S. Kollaopoulos, S. Rao, A nearly linear-time approximation scheme for the Euclidean k -median problem, in *Proc. of ESA* (1999), pp. 378–389
29. M.R. Korupolu, C.G. Plaxton, R. Rajaraman, Analysis of a local search heuristic for facility location problems. *J. Algorithms* **37**(1), 146–188 (2000)
30. J.H. Lin, J.S. Vitter, Approximation algorithms for geometric median problems. *Inf. Process. Lett.* **44**, 245–249 (1992)
31. R. Mettu, C.G. Plaxton, Optimal time bounds for approximate clustering, in *Proc. of UAI* (2002), pp. 344–351
32. R. Mettu, C.G. Plaxton, The online median problem. *SIAM J. Comput.* **32**(3) (2003)

33. A. Meyerson, L. O'Callaghan, S. Plotkin, A k -median algorithm with running time independent of data size. *Mach. Learn.* **56**, 61–87 (2004)
34. N. Mishra, D. Oblinger, L. Pitt, Sublinear time approximate clustering, in *Proc. of SODA* (2001)
35. M. Thorup, Quick k -median, k -center, and facility location for sparse graphs, in *Proc. of ICALP* (2001), pp. 249–260
36. V. Vazirani, *Approximation Algorithms* (Springer, Berlin, 2001)
37. A. Weber, *Über den Standort der Industrien (Theory of the Location of Industries)* (University of Chicago Press, Chicago, 1929). Translated in 1929 by Carl J. Friedrich from Weber's 1909 text
38. T. Zhang, R. Ramakrishnan, M.L. Birch, An efficient data clustering method for very large databases, in *Proc. of SIGMOD* (1996), pp. 103–114

Mining Decision Trees from Streams

Geoff Hulten and Pedro Domingos

1 Introduction

The massive data streams that occur in many domains today are a tremendous opportunity for data mining, but also pose a difficult challenge. Open-ended, high-volume streams potentially allow us to build far richer and more detailed models than before. On the other hand, a system capable of mining streaming data as fast as it arrives must meet a number of stringent design criteria:

- It must require small constant time per record, otherwise it will inevitably fall behind the data, sooner or later.
- It must use only a fixed amount of main memory, irrespective of the total number of records it has seen.
- It must be able to build a model using at most one sequential scan of the data, since it may not have time to revisit old records, or disk space to store them.
- It must make a usable model available at any point in time, as opposed to only when it is done processing the data, since it may never be done processing.
- Ideally, it should produce a model that is equivalent (or nearly identical) to the one that would be obtained by the corresponding ordinary database mining algorithm, operating without the above constraints.

We have developed a general framework for transforming data mining algorithms into stream mining ones that satisfy these criteria [12]. The basic idea in our approach is to, at each step of the algorithm, use the minimum number of examples

G. Hulten
Microsoft Research, Redmond, WA, USA
e-mail: ghulten@microsoft.com

P. Domingos (✉)
Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA
e-mail: pedrod@cs.washington.edu

from the stream that (with high probability) produces approximately the same results as would be obtained with infinite data. We have applied this framework to learning Bayesian network structure [12], k -means clustering [4], the EM algorithm for mixtures of Gaussians [5], learning relational models [14], and learning in time-changing domains [15]. In this chapter, we illustrate the use of our framework by applying it to what is perhaps the most widely used form of data mining: decision tree induction.

In Sect. 2, we present VFDT (Very Fast Decision Tree learner), our algorithm for learning decision trees from massive data streams, and give theoretical bounds on the quality of the trees it produces. In Sect. 3, we empirically evaluate VFDT on a large collection of synthetic data sets. In Sect. 4 we apply VFDT to the task of improving Web page caching. We conclude with a brief discussion of related work.

2 The VFDT System

The classification problem is generally defined as follows. A set of N training examples of the form (\mathbf{x}, y) is given, where y is a discrete class label and \mathbf{x} is a vector of d attributes, each of which may be symbolic or numeric. The goal is to produce from these examples a model $y = f(\mathbf{x})$ that will predict the classes y of future examples \mathbf{x} with high accuracy. For example, \mathbf{x} could be a description of a client's recent purchases, and y the decision to send that customer a catalog or not; or \mathbf{x} could be a record of a cellular-telephone call, and y the decision whether it is fraudulent or not. Decision trees address this problem as follows. Each internal node in the tree contains a test on an attribute, each branch from a node corresponds to a possible outcome of the test, and each leaf contains a class prediction. The label $y = DT(\mathbf{x})$ for an example \mathbf{x} is obtained by passing the example down from the root to a leaf, testing the appropriate attribute at each node and following the branch corresponding to the attribute's value in the example.

A decision tree is learned by recursively replacing leaves by test nodes, starting at the root. The attribute to test at a node is chosen by comparing all the available attributes and choosing the best one according to some heuristic measure. Classic decision tree learners like ID3, C4.5 and CART [1, 19] assume that all training examples can be stored simultaneously in main memory, and are thus severely limited in the number of examples they can learn from. Disk-based decision tree learners like SLIQ [17] and SPRINT [21] assume the examples are stored on disk, and learn by repeatedly reading them in sequentially (effectively once per level in the tree). While this greatly increases the size of usable training sets, it can become prohibitively expensive when learning complex trees (i.e., trees with many levels), and fails when data sets are too large to fit in the available disk space or when data is arriving on an open-ended stream.

VFDT learns from massive open-ended data streams by noting with Catlett [2] and others [8, 18] that, in order to find the best attribute to test at a given node, it may be sufficient to consider only a small subset of the training examples that

are relevant to that node. Given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate tests there, and so on recursively. We solve the difficult problem of deciding exactly how many examples are necessary at each node by using a statistical result known as the *Hoeffding bound* (or additive Chernoff bound) [11]. Consider a real-valued random variable r whose range is R . Suppose we have made n independent observations of this variable, and computed their mean \bar{r} . One form of Hoeffding bound states that, with probability $1 - \delta$, the true mean of the variable is at least $\bar{r} - \epsilon$, where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}. \quad (1)$$

The Hoeffding bound has the very attractive property that it is independent of the probability distribution generating the observations. The price of this generality is that the bound is more conservative than the distribution-dependent ones (i.e., it will take more observations to reach the same δ and ϵ). Let $G(\cdot)$ be the heuristic measure used to choose attributes (e.g., the measure could be information gain as in C4.5, or the Gini index as in CART). Our goal is to ensure that, with high probability, the attribute chosen using n examples (where n is as small as possible) is the same that would be chosen using infinite examples. Assume G is to be maximized, let X_a be the attribute with highest observed \bar{G} after seeing n examples, and X_b be the second-best. Let $\Delta\bar{G} = \bar{G}(X_a) - \bar{G}(X_b) \geq 0$ be the difference between their observed heuristic values. Then, given a desired δ , the Hoeffding bound guarantees that X_a is the correct choice with probability $1 - \delta$ if n examples have been seen at this node and $\Delta\bar{G} > \epsilon$. Thus a node needs to accumulate examples from the stream until ϵ becomes smaller than $\Delta\bar{G}$. At this point the node can be split using the current best attribute, and succeeding examples passed to the new leaves. The only case in which this procedure does not yield a decision in finite time occurs when ΔG is zero (or very close to it). In this case there is no n that will suffice to make $\Delta\bar{G} > \epsilon$. However, in this case we do not care which attribute is selected because their performance on the metric we are interested in is the same. If we stipulate a minimum difference τ below which we are indifferent, the procedure above is guaranteed to terminate after seeing at most $n = \lceil \frac{1}{2}(R/\tau)^2 \ln(1/\delta) \rceil$ examples. In other words, the time required to choose an attribute is constant, independent of the size of the data stream. This leads to the VFDT algorithm, shown in pseudocode in Table 1. The pseudocode shown is only for discrete attributes; it can be extended to numeric ones with the usual methods. The sequence of examples T may be infinite, in which case the procedure never terminates, and at any point in time a parallel procedure can use the current tree DT to make class predictions.

The most significant part of the time cost per example is recomputing G . VFDT can reduce this cost by only checking for a winning attribute once for every Δn examples that arrive at a leaf. As long as VFDT processes examples faster than they arrive, which will be the case in all but the most demanding applications, the sole obstacle to learning arbitrarily complex models will be the finite RAM available.

Table 1 The VFDT algorithm

Inputs: A stream of iid samples $T = \{X_1, X_2, \dots, X_\infty\}$,
X, a set of discrete attributes,
 $G(\cdot)$, a split evaluation function
 δ , one minus the desired probability of choosing the correct attribute at
any given node,
 τ , a user specified tie threshold,
 f , a bound function (we use the Hoeffding bound, see Eq. (1)).

Output: DT , a decision tree.

Procedure VFDT ($T, \mathbf{X}, G, \delta, \tau, f$)

/* Initialize */

Let DT be a tree with a single leaf l_1 (the root).For each class y_k For each value x_{ij} of each attribute $X_i \in \mathbf{X}$ Let $n_{ijk}(l_1) = 0$. /* The sufficient statistics needed to calculate G at l */

/* Process the examples */

For each example (\mathbf{x}, y) in T Sort (\mathbf{x}, y) into a leaf l using DT .For each x_{ij} in \mathbf{x} such that $X_i \in \mathbf{X}_l$ Increment $n_{ijk}(l)$.Label l with the majority class among the examples seen so far at l .If the examples seen so far at l are not all of the same class, thenCompute $\overline{G}(X_l)$ for each attribute $X_i \in \mathbf{X}_l$ using the counts $n_{ijk}(l)$.Let X_a be the attribute with highest \overline{G} .Let X_b be the attribute with the second highest \overline{G} .Compute ϵ using $f(\cdot)$.If $\overline{G}(X_a) - \overline{G}(X_b) > \epsilon$ or $\epsilon < \tau$, thenReplace l by an internal node that splits on X_a .

For each branch of the split

Add a new leaf l_m , and let $\mathbf{X}_m = \mathbf{X} - \{X_a\}$.For each class y_k and each value x_{ij} of each attribute $X_i \in \mathbf{X}_m$ Let $n_{ijk}(l_m) = 0$.Return DT .

VFDT's memory use is dominated by the memory required to keep sufficient statistics for all growing leaves. If d is the number of attributes, v is the maximum number of values per attribute, and c is the number of classes, VFDT requires $O(dvc)$ memory to store the necessary counts at each leaf. If l is the number of leaves in the tree, the total memory required is $O(l dvc)$. This is independent of the number of examples seen, if the size of the tree depends only on the "true" concept and is independent of the size of the training set. If VFDT ever exceeds the RAM available, it temporarily deactivates learning at some leaves. In particular, if p_l is the probability that an arbitrary example will fall into leaf l , and e_l is the observed error rate at that leaf on the training data that reaches it, then $p_l e_l$ is an estimate of the maximum error reduction achievable by refining the leaf. When available RAM is exhausted, we disable the leaves with the lowest values for $P_l e_l$. When a leaf is deactivated, the memory it was using for its sufficient statistics is freed. A leaf can then be re-

activated if it becomes more promising than a currently active leaf. VFDT caches training examples in RAM in order to make best use of all available memory in the early part of a run (before RAM is filled by sufficient statistics). This is similar to what batch learners do, and it allows VFDT to reuse training examples for decisions on several levels of the induced tree.

2.1 Quality Guarantees

A key property of the VFDT algorithm is that it is possible to guarantee under realistic assumptions that the trees produced by the algorithm (as long as no decisions are made by reaching the ‘indifference threshold’ $n = \lceil \frac{1}{2}(R/\tau)^2 \ln(1/\delta) \rceil$) are asymptotically arbitrarily close to the ones produced by a batch learner (i.e., a learner that uses all the examples to choose a test at each node). In other words, the incremental nature of the VFDT algorithm does not significantly affect the quality of the trees it produces. In order to make this statement precise, we need to define the notion of *disagreement* between two decision trees. Let $P(\mathbf{x})$ be the probability that the attribute vector (loosely, example) \mathbf{x} will be observed, and let $I(\cdot)$ be the indicator function, which returns 1 if its argument is true and 0 otherwise.

Definition 1 The *extensional disagreement* Δ_e between two decision trees DT_1 and DT_2 is the probability that they will produce different class predictions for an example,

$$\Delta_e(DT_1, DT_2) = \sum_{\mathbf{x}} P(\mathbf{x}) I[DT_1(\mathbf{x}) \neq DT_2(\mathbf{x})].$$

Consider that two internal nodes are different if they contain different tests, two leaves are different if they contain different class predictions, and an internal node is different from a leaf. Consider also that two paths through trees are different if they differ in length or in at least one node.

Definition 2 The *intensional disagreement* Δ_i between two decision trees DT_1 and DT_2 is the probability that the path of an example through DT_1 will differ from its path through DT_2 ,

$$\Delta_i(DT_1, DT_2) = \sum_{\mathbf{x}} P(\mathbf{x}) I[\text{Path}_1(\mathbf{x}) \neq \text{Path}_2(\mathbf{x})]$$

where $\text{Path}_i(\mathbf{x})$ is the path of example \mathbf{x} through tree DT_i .

Two decision trees agree intensionally on an example iff they are indistinguishable for that example: the example is passed down exactly the same sequence of nodes, and receives an identical class prediction. Intensional disagreement is a stronger notion than extensional disagreement, in the sense that $\forall_{DT_1, DT_2} \Delta_i(DT_1, DT_2) \geq \Delta_e(DT_1, DT_2)$.

Let p_l be the probability that an example that reaches level l in a decision tree falls into a leaf at that level. To simplify, we will assume that this probability is constant, i.e., $\forall_l p_l = p$, where p will be termed the *leaf probability*. This is a realistic assumption, in the sense that it is typically approximately true for the decision trees that are generated in practice. Let $VFDT_\delta$ be the tree produced by the VFDT algorithm with desired probability δ given an infinite sequence of examples T , and DT_* be the asymptotic batch decision tree induced by choosing at each node the attribute with true greatest G (i.e., by using infinite examples at each node). Let $E[\Delta_i(VFDT_\delta, DT_*)]$ be the expected value of $\Delta_i(VFDT_\delta, DT_*)$, taken over all possible infinite training sequences. We can then state the following result.

Theorem 1 *If $VFDT_\delta$ is the tree produced by the VFDT algorithm with desired probability δ given infinite examples (Table 1), DT_* is the asymptotic batch tree, and p is the leaf probability, then $E[\Delta_i(VFDT_\delta, DT_*)] \leq \delta/p$.*

Proof For brevity, we will refer to intensional disagreement simply as disagreement. Consider an example \mathbf{x} that falls into a leaf at level l_v in $VFDT_\delta$, and into a leaf at level l_d in DT_* . Let $l = \min\{l_v, l_d\}$. Let $\text{Path}_V(\mathbf{x}) = (N_1^V(\mathbf{x}), N_2^V(\mathbf{x}), \dots, N_l^V(\mathbf{x}))$ be \mathbf{x} 's path through $VFDT_\delta$ up to level l , where $N_i^V(\mathbf{x})$ is the node that \mathbf{x} goes through at level i in $VFDT_\delta$, and similarly for $\text{Path}_D(\mathbf{x})$, \mathbf{x} 's path through DT_* . If $l = l_h$ then $N_l^V(\mathbf{x})$ is a leaf with a class prediction, and similarly for $N_l^D(\mathbf{x})$ if $l = l_d$. Let I_i represent the proposition “ $\text{Path}_V(\mathbf{x}) = \text{Path}_D(\mathbf{x})$ up to and including level i ,” with $I_0 = \text{True}$. Notice that $P(l_v \neq l_d)$ is included in $P(N_l^H(\mathbf{x}) \neq N_l^D(\mathbf{x}) | I_{l-1})$, because if the two paths have different lengths then one tree must have a leaf where the other has an internal node. Then, omitting the dependency of the nodes on \mathbf{x} for brevity,

$$\begin{aligned} & P(\text{Path}_V(\mathbf{x}) \neq \text{Path}_D(\mathbf{x})) \\ &= P(N_1^V \neq N_1^D \vee N_2^V \neq N_2^D \vee \dots \vee N_l^V \neq N_l^D) \\ &= P(N_1^V \neq N_1^D | I_0) + P(N_2^V \neq N_2^D | I_1) + \dots + P(N_l^V \neq N_l^D | I_{l-1}) \\ &= \sum_{i=1}^l P(N_i^V \neq N_i^D | I_{i-1}) \leq \sum_{i=1}^l \delta = \delta l. \end{aligned} \tag{2}$$

Let $VFDT_\delta(T)$ be the VFDT tree generated from training sequence T . Then $E[\Delta_i(DT_\delta, DT_*)]$ is the average over all infinite training sequences T of the probability that an example's path through $VFDT_\delta(T)$ will differ from its path through DT_* :

$$\begin{aligned} & E[\Delta_i(VFDT_\delta, DT_*)] \\ &= \sum_T P(T) \sum_{\mathbf{x}} P(\mathbf{x}) I[\text{Path}_V(\mathbf{x}) \neq \text{Path}_D(\mathbf{x})] \\ &= \sum_{\mathbf{x}} P(\mathbf{x}) P(\text{Path}_V(\mathbf{x}) \neq \text{Path}_D(\mathbf{x})) \\ &= \sum_{i=1}^{\infty} \sum_{\mathbf{x} \in L_i} P(\mathbf{x}) P(\text{Path}_V(\mathbf{x}) \neq \text{Path}_D(\mathbf{x})) \end{aligned} \tag{3}$$

where L_i is the set of examples that fall into a leaf of DT_* at level i . According to Eq. (2), the probability that an example's path through $VFDT_\delta(T)$ will differ from its path through DT_* , given that the latter is of length i , is at most δi (since $i \geq l$). Thus

$$E[\Delta_i(VFDT_\delta, DT_*)] \leq \sum_{i=1}^{\infty} \sum_{\mathbf{x} \in L_i} P(\mathbf{x})(\delta i) = \sum_{i=1}^{\infty} (\delta i) \sum_{\mathbf{x} \in L_i} P(\mathbf{x}). \quad (4)$$

The sum $\sum_{\mathbf{x} \in L_i} P(\mathbf{x})$ is the probability that an example \mathbf{x} will fall into a leaf of DT_* at level i , and is equal to $(1 - p)^{i-1} p$, where p is the leaf probability. Therefore,

$$\begin{aligned} E[\Delta_i(VFDT_\delta, DT_*)] &\leq \sum_{i=1}^{\infty} (\delta i)(1 - p)^{i-1} p = \delta p \sum_{i=1}^{\infty} i(1 - p)^{i-1} \\ &= \delta p \left[\sum_{i=1}^{\infty} (1 - p)^{i-1} + \sum_{i=2}^{\infty} (1 - p)^{i-1} + \cdots + \sum_{i=k}^{\infty} (1 - p)^{i-1} + \cdots \right] \\ &= \delta p \left[\frac{1}{p} + \frac{1-p}{p} + \cdots + \frac{(1-p)^{k-1}}{p} + \cdots \right] \\ &= \delta [1 + (1 - p) + \cdots + (1 - p)^{k-1} + \cdots] = \delta \sum_{i=0}^{\infty} (1 - p)^i = \frac{\delta}{p}. \end{aligned} \quad (5)$$

This completes the demonstration of Theorem 1. \square

An immediate corollary of Theorem 1 is that the expected extensional disagreement between $VFDT_\delta$ and DT_* is also asymptotically at most δ/p (although in this case the bound is much looser). Another corollary is that there exists a subtree of the asymptotic batch tree such that the expected disagreement between it and the tree learned by VFDT on finite data is at most δ/p . In other words, if δ/p is small then VFDT's tree learned on finite data is very similar to a subtree of the asymptotic batch tree. A useful application of Theorem 1 is that, instead of δ , users can now specify as input to the VFDT algorithm the maximum expected disagreement they are willing to accept, given enough examples for the tree to settle. The latter is much more meaningful, and can be intuitively specified without understanding the workings of the algorithm or the Hoeffding bound. The algorithm will also need an estimate of p , which can easily be obtained (for example) by running a conventional decision tree learner on a manageable subset of the data.

3 Empirical Evaluation of VFDT on Synthetic Data

A system like VFDT is only useful if it is able to learn more accurate trees than a conventional system, given similar computational resources. It should be able to

use the examples that are beyond a conventional system’s ability to process to learn better models. In this section, we test this empirically by comparing VFDT with C4.5 release 8 [19], the incremental decision tree learner ITI [23], and our implementation of SPRINT [21] on a series of synthetic data streams. Using synthetic data streams allows us to freely vary the relevant parameters of the learning process and more fully explore the capabilities of our system. In Sect. 4, we describe an evaluation on a real-world data set.

We first describe the data streams used for our experiments. They were created by randomly generating decision trees and then using these trees to assign classes to randomly generated examples. We produced the random decision trees by starting from a tree with a single leaf node (the root) and repeatedly replacing leaf nodes with nodes that tested randomly selected attributes. After the first three levels of the tree each selected leaf had a probability of f of being pre-pruned instead of replaced by a split. Any branch that reached a depth of 18 was pruned at that depth. Class labels were assigned to leaves randomly (with uniform probability). We then generated a stream of 50 million training examples for each tree by sampling uniformly from the instance space, assigning classes to the examples according to the corresponding synthetic tree, and adding class and attribute noise by flipping each with probability n . We also generated 50,000 separate testing examples for each concept using the same procedure. We used nine parameter settings to generate trees: every combination of f in {12 %, 17 %, 23 %} and n in {5 %, 10 %, 15 %}. We also used six different random seeds for each of these settings to produce a total of 54 data streams. On average the trees we generated using this method contained about 100,000 nodes. Every domain we experimented with had two classes and 100 binary attributes. We ran our experiments on a cluster of five Pentium III/1 GHz machines with 1 GB of RAM, all running Linux. We reserved nine of the data streams, one from each of the parameter settings above, to tune the algorithms’ parameters.

We will now briefly describe the learning algorithms used in our experiments:

- C4.5—The standard program for learning decision trees from data that fits in RAM. We used the reserved data streams to tune C4.5’s ‘-c’ (prune confidence level) parameter, and found 5 % to achieve the best results.
- ITI—An incremental decision tree learner for data sets that fit in RAM. We tuned ITI’s ‘-P’ parameter, which controls the complexity of the trees ITI learns. This parameter specifies a minimum number of examples of the second most common class that must be present in every leaf. We found 50 to give the best results.
- SPRINT—An algorithm designed to learn decision trees from data sets that can fit on disk, but are too large to load into RAM. We used our own implementation of SPRINT for these experiments, which shares code with our VFDT implementation. SPRINT pruned its trees using the same code as VFDT’s post-prune option, and both algorithms used the same pruning criteria: reserving 5 % of training data for pruning. Once pruning was fixed, our implementation of SPRINT had no tunable parameters.
- VFDT—We tried three forms of pruning: no-pruning, pre-pruning (no split is made unless it improves $G(\cdot)$ by at least 0.005), and post-pruning (each training sample from the stream was reserved with 5 % probability for pruning until a

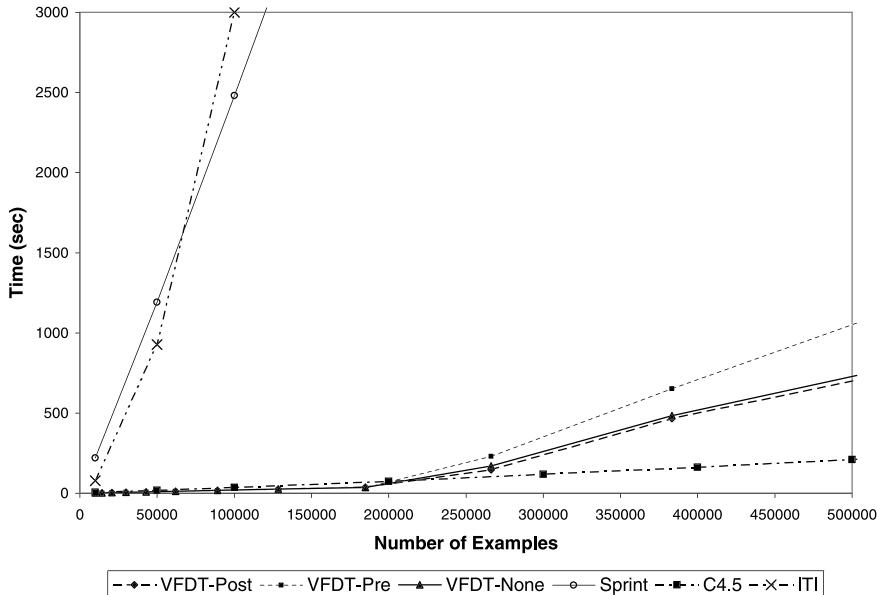


Fig. 1 Average runtimes (zoom)

maximum of 2 million examples were reserved; a copy of the tree was pruned with this data using standard reduced error pruning before generating each point in the following graphs). We used a Δn (the number examples to accumulate before checking for a winner) of 1,000 for all experiments. We used the reserved data streams to tune VFDT's two remaining parameters, τ and δ . We found the best setting for post-pruning to be $\tau = 0.05, \delta = 1^{-7}$; the best for no-pruning to be $\tau = 0.1, \delta = 1^{-9}$; and the best for pre-pruning to be $\tau = 0.1, \delta = 1^{-5}$.

The incremental systems, VFDT and ITI, were allowed to learn from the streams directly. We ran ITI on the first 100,000 examples in each stream, because it was too slow to learn from more. We ran VFDT on the entire streams and limited its dynamic memory allocations to 600 MB (the rest of the memory on our computers was reserved for the data generating process and other system processes). For the batch systems, C4.5 and SPRINT, we sampled several data sets of increasing size from the data streams and ran the learners on each of them in turn. C4.5 was run on data sets up to 800 MB in size, which contained 2 million samples. We allowed SPRINT to use up to 1.6 GB of data, or 4 million samples (about as much data as it can process in the time it takes VFDT to process the entire 50 million samples).

Figures 1 and 2 show the runtime of the algorithms averaged over all 45 of the data streams. The runtimes of all the systems seem to increase linearly with additional data. ITI and SPRINT were the slower of the systems by far; they were approximately two orders of magnitude slower than C4.5 and VFDT. ITI was slower because of the cost of incrementally keeping its model up-to-date as every example arrives, which in the worse case requires rebuilding the entire tree. This suggests that

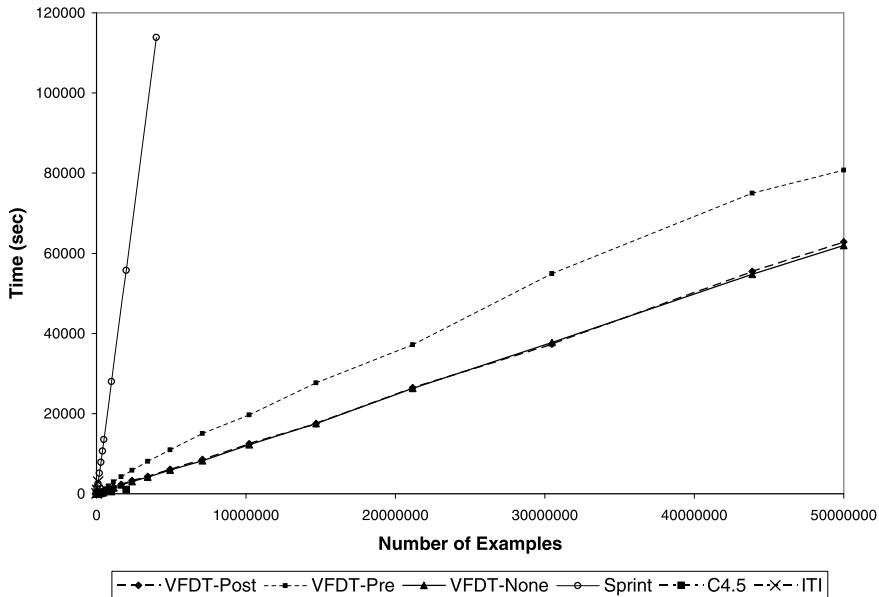


Fig. 2 Average runtimes

such incremental maintenance is not viable for massive data streams. SPRINT's was slower because it needed to do a great deal of additional I/O to scan the entire training set once per level of the tree it learned. VFDT was approximately 3 times slower than C4.5 (up to the point where C4.5 was not able to run on more data because it ran out of RAM). VFDT used this additional time to: track its memory allocations; swap searches into and out of memory; and manage example caches. Remember, however, that C4.5 needed to be run once for every test point, while VFDT was able to incrementally incorporate examples, and produce all test points with a single pass over each of the data streams. Surprisingly, VFDT-pre was the slowest of the VFDT algorithms. Looking deeper, we found that it learned smaller trees, usually had a larger fraction of its searches active in RAM at a time, and thus spent more time updating sufficient statistics and checking for winning searches than the other VFDT variants.

Next, we compared the error rates of the models learned by the systems and Fig. 3 contains results. The final error rate achieved by each of the three VFDT systems was better than the final error rate achieved by any of the non-VFDT systems. This demonstrates that VFDT is able to take advantage of examples past those that could be used by the other systems to learn better models. It is also interesting to note that one of the VFDT systems had the best—or nearly the best—average performance after any number of examples: VFDT-Pre with less than 100,000 examples, VFDT-None from there to approximately 5 million examples; and VFDT-Post from there until the end at 50 million examples. We also compared the systems on their ability to quickly learn models with low error rate. Figure 4 shows the error rate the algo-

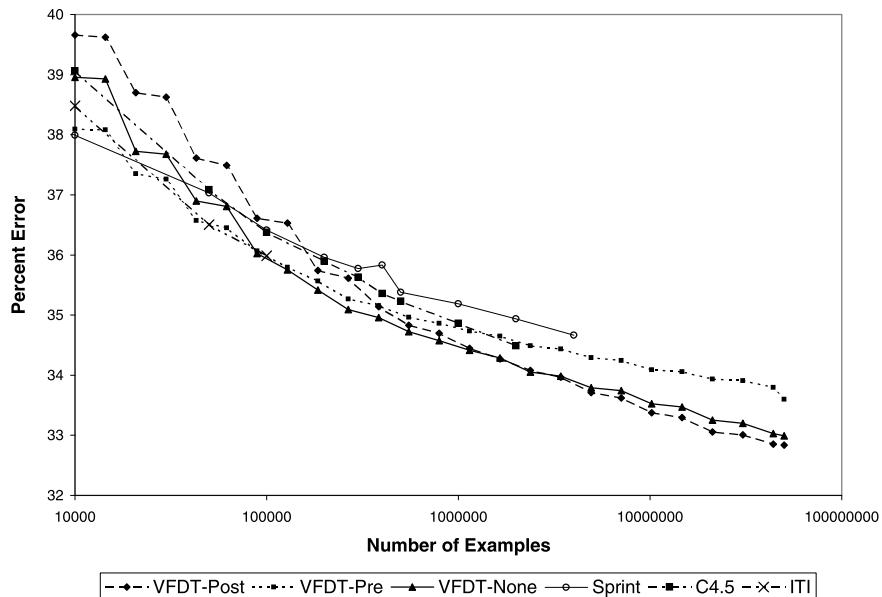


Fig. 3 Average test set error rate vs. number of training examples

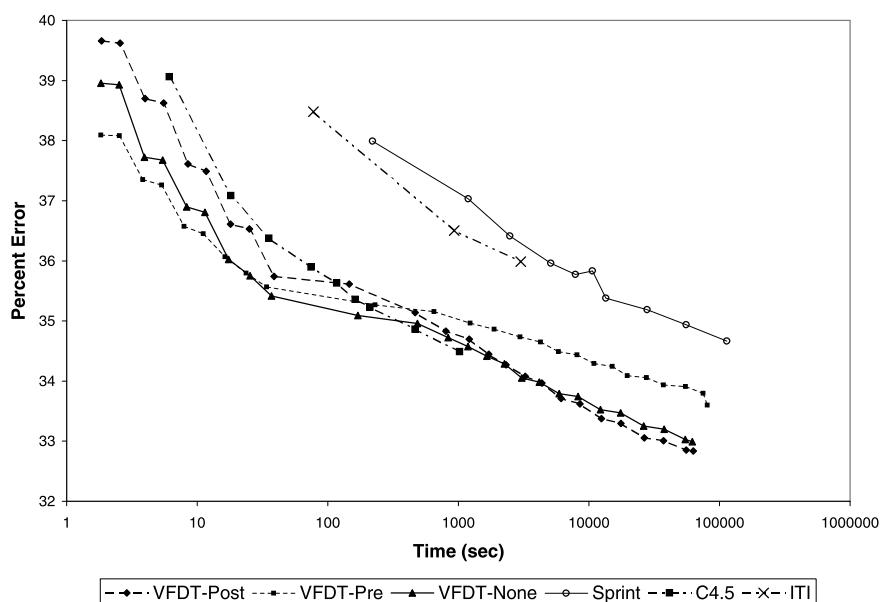


Fig. 4 Average test set error rate vs. time

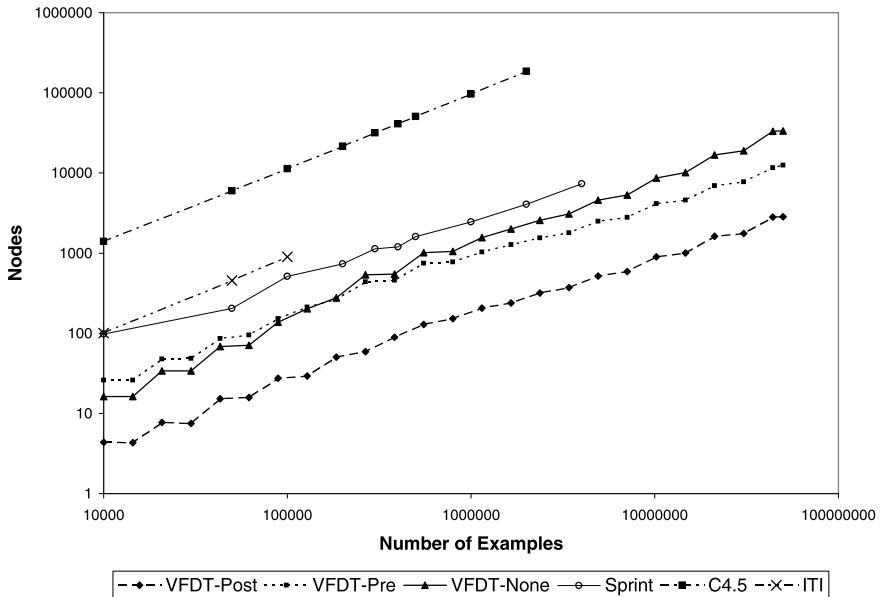
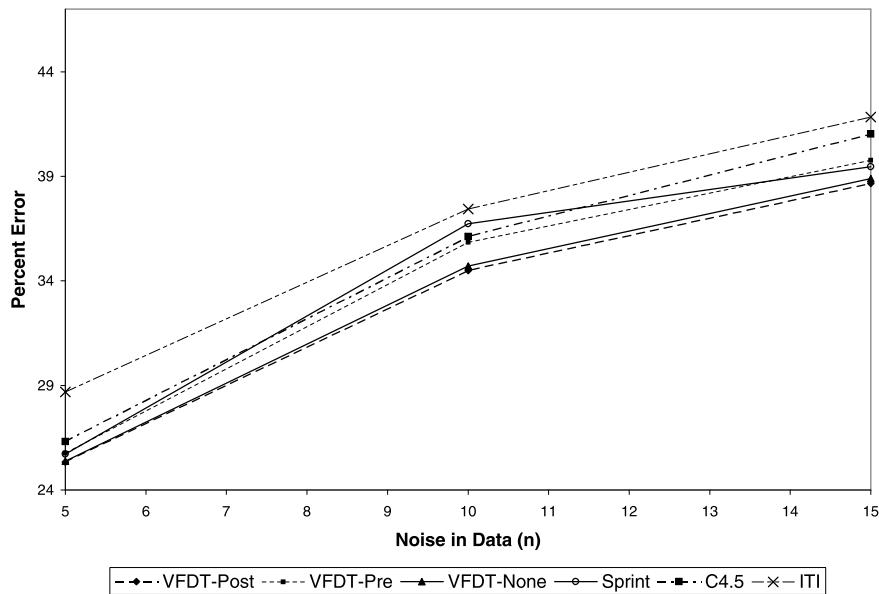
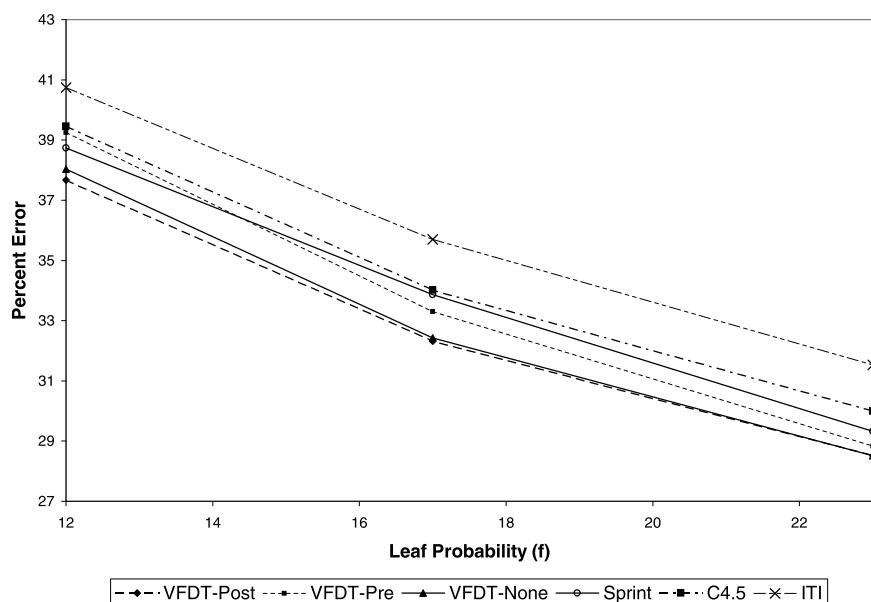


Fig. 5 Average number of nodes induced by the algorithms

rithms achieved on the y -axis and the time the algorithms took to achieve it on the x -axis, it thus shows which algorithm is able to most quickly produce high quality results. Notice that one of the VFDT settings had the best error rate after nearly any amount of time, and that ITI and SPRINT were substantially less efficient than VFDT according this metric.

Figure 5 shows the average number of nodes in the models learned by each of the algorithms. The number of nodes learned seemed to scale approximately linear with additional training examples for all of the systems. C4.5 learned the most nodes of any of the systems by several orders of magnitude; and VFDT-Post learned the fewest nodes by an order of magnitude. Notice that SPRINT learned many more nodes than VFDT-Post even though they used the same pruning implementation. This occurred because SPRINT's model contained every split that had gain on the training data, while the splits in VFDT-Post's tree were chosen, with high confidence, by our method. VFDT's simultaneous advantage with error rate and concept size is additional evidence that our method for selecting splits is effective.

Figures 6 and 7 show how the algorithms' performance varied with changes to the amount of noise added (n) and to the size of the target concepts (f). Each data point represents the average best error rate achieved by the respective algorithm on all the runs with the indicated n and f . VFDT-Post had the best performance in every setting, followed closely by VFDT-None. VFDT-Pre's error rate was slightly higher than SPRINT's on the runs with the highest noise level and the runs with the largest concepts. These results suggest that VFDT's performance scales gracefully as concept size or noise increases.

**Fig. 6** Average performance by noise level**Fig. 7** Average performance by concept size

We calculated the rate at which VFDT is able to incorporate training examples to determine how large a data stream it is able to keep up with in real time. VFDT-None was able to learn from an average of 807 examples per second. Extrapolating from this, we estimate it is able to learn from approximately 70 million examples per day. Remember that these experiments were conducted on 1 GHz Pentium III processors—hardware that is already modest by current standards. Further, if faced with a faster data stream VFDT can be accelerated by increasing Δn or by more aggressively disabling learning at unpromising leaves. For example, using a version of VFDT that did not cache examples or reactivate disabled leaves, we were able to mine a stream of one billion examples in about one day.

In summary, our experiments demonstrate the utility of a tool like VFDT in particular and of our method for mining high-speed data streams in general: VFDT achieves the same results as the traditional system, in an order of magnitude less time; VFDT is then able to use additional time and data to learn more accurate models, while traditional system's memory requirements keep them from learning on more data; finally, VFDT's incremental nature allows it to smoothly refine its model as more data arrives, while batch systems must be rerun from scratch to incorporate the latest data.

4 Application of VFDT to Web Data

We further evaluated VFDT by using it to mine the stream of Web page requests emanating from the whole University of Washington main campus. The nature of the data is described in detail in [25]. In our experiments, we used a one-week anonymized trace of all the external Web accesses made from the university campus in May 1999. There were 23,000 active clients during this one-week trace period, and the entire university population is estimated at 50,000 people (students, faculty and staff). The trace contains 82.8 million requests, which arrive at a peak rate of 17,400 per minute. The size of the compressed trace file is about 20 GB. Each request in the log is tagged with an anonymized organization ID that associates the request with one of the 170 organizations (colleges, departments, etc.) within the university. One purpose this data can be used for is to improve Web caching. The key to this is predicting as accurately as possible which hosts and pages will be requested in the near future, given recent requests. We applied decision tree learning to this problem in the following manner. We split the campus-wide request log into a series of equal time slices $T_0, T_1, \dots, T_t, \dots$. We used time slices of 30 seconds, 1 minute, 5 minutes, and 15 minutes for our experiments. Let N_T be the number of time slices in a day. For each organization $O_1, O_2, \dots, O_i, \dots, O_{170}$ and each of the 244k hosts appearing in the logs $H_1, \dots, H_j, \dots, H_{244k}$, we maintain a count of how many times the organization accessed the host in the time slice, C_{ijt} . Then for each time slice and host accessed in that time slice (T_t, H_j) we generate an example with attributes: $C_{1,jt}, \dots, C_{ijt}, \dots, C_{170,jt}$, $t \bmod N_T$, and class 1 if any request is made to H_j in time slice T_{t+1} and 0 if not. This can be carried out in real time using

Table 2 Results of the Web experiments. ‘Match Time’ is the time it took VFDT to match C4.5’s best accuracy. ‘VFDT Time’ is the time it took VFDT to do six complete scans of the training set

Data set	C4.5 Error (%)	VFDT Error (%)	C4.5 Time (s)	Match Time (s)	VFDT Time (s)
30 s	37.70	36.95 %	60k	5k	247k
1 min	37.30	36.67 %	71k	6k	160k
5 min	33.59	33.23 %	61k	15k	72k

modest resources by keeping statistics on the last and current time slices C_{t-1} and C_t in memory, only keeping counts for hosts that actually appear in a time slice (we never needed more than 30k counts), and outputting the examples for C_{t-1} as soon as C_t is complete. Testing was carried out on the examples from the last day in each of the data sets. The ‘30 s’ data set had 10,850k training examples, 1,944k testing examples and 46.0 % were class 0. The ‘1 min’ data set had 7,921k training examples, 1,419k testing examples and 45.4 % were class 0. The ‘5 min’ data set had 3,880k training examples, 696k testing examples and 52.5 % were class 0. The ‘15 min’ data set had 2,555k training examples, 454k testing examples, and 58.1 % were class 0.

We ran C4.5 and VFDT on these data sets, on a 1 GHz machine with 1 GB of RAM running Linux. We tested C4.5 on data sets up to 1.2M examples (roughly 800 MB; the remaining RAM was reserved for C4.5’s learned tree and system processes). C4.5’s data sets were created by random sampling from the available training data. Notice that the 1.2M examples C4.5 was able to process is less than a day’s worth for the ‘30 s’ and ‘1 min’ data sets and less than 50 % of the training data in the remaining cases. We allowed VFDT to do six passes over the training data—and thus incorporate each training example in its tree at six different places—and we allowed it to use 400 MB of RAM. We tuned the algorithms’ parameters on the ‘15 min’ data set. C4.5’s best prune confidence was 5 %, and VFDT’s best parameters were: no-pruning, $\delta = 1^{-7}$, $\tau = 0.1$, and $\Delta_n = 300$. We learned from numeric attributes by maintaining a sorted list of the values seen and considering a split between each pair of adjacent values with different classes. There were very few distinct values for each attribute so this method did not use much time or RAM.

Table 2 summarizes the results of these experiments. VFDT was able to learn a more accurate model than C4.5 on every data set. VFDT’s improvement over C4.5 was largest on the ‘30 s’ data set and was smallest on the ‘5 min’ data set. Notice that C4.5 was able to use about 11 % of the ‘30 s’ data set and about 31 % of the ‘5 min’ data set. This supports the notion that VFDT’s performance improvement over a traditional system will depend on the amount of additional data that it is able to process past the point where the traditional system runs into a resource limit. VFDT was also much faster than C4.5 in the following sense. We noted the accuracy of the best model learned by C4.5 and the time it took it to learn it. We then examined VFDT’s results to determine how long it took it to learn a model with that same accuracy. We found that VFDT achieved C4.5’s best accuracy an order of magnitude faster on the ‘30 s’ and ‘1 min’ data sets and 4 times faster on the ‘5 min’ data set. Figures 8, 9, and 10 contain a more detailed picture of VFDT’s simultaneous speed

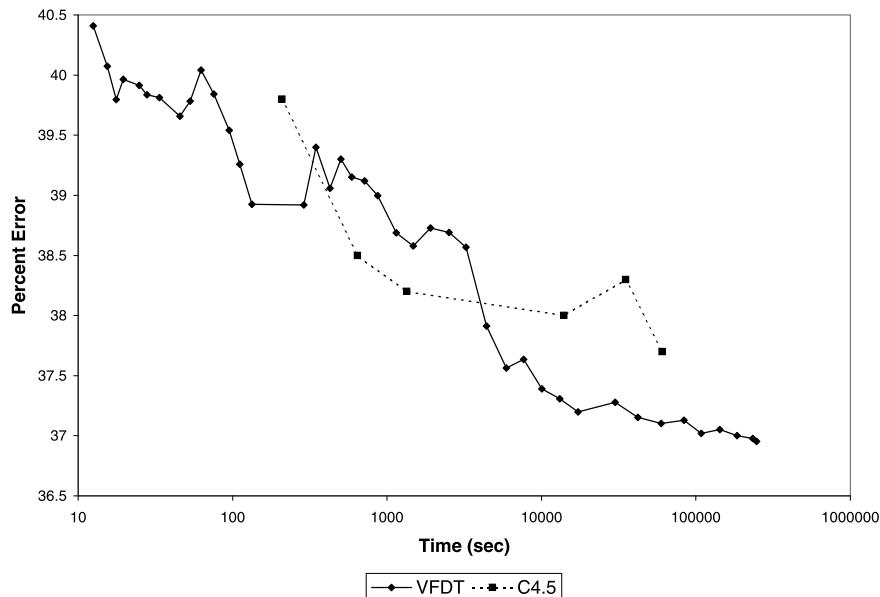


Fig. 8 Time vs. Error Rate on the '30 s' data set

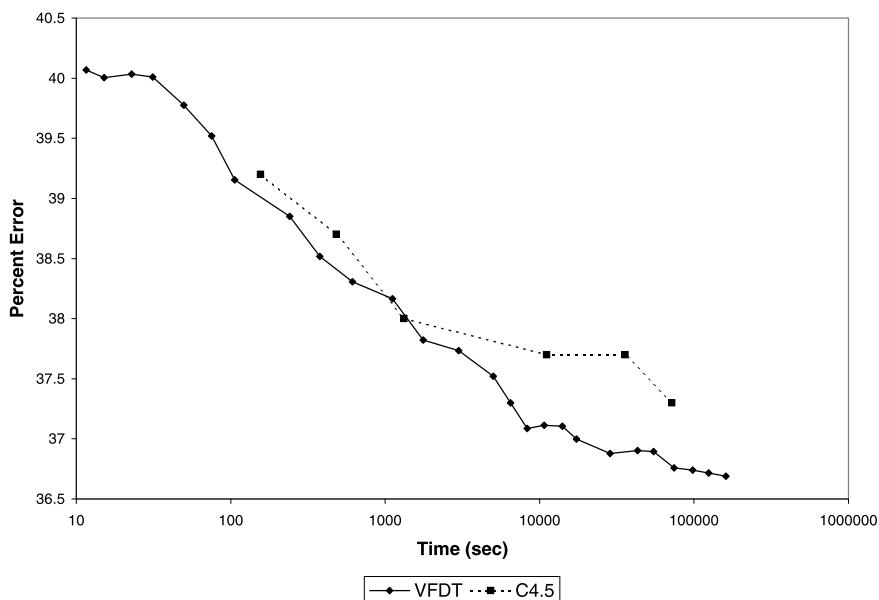


Fig. 9 Time vs. Error Rate on the '1 min' data set

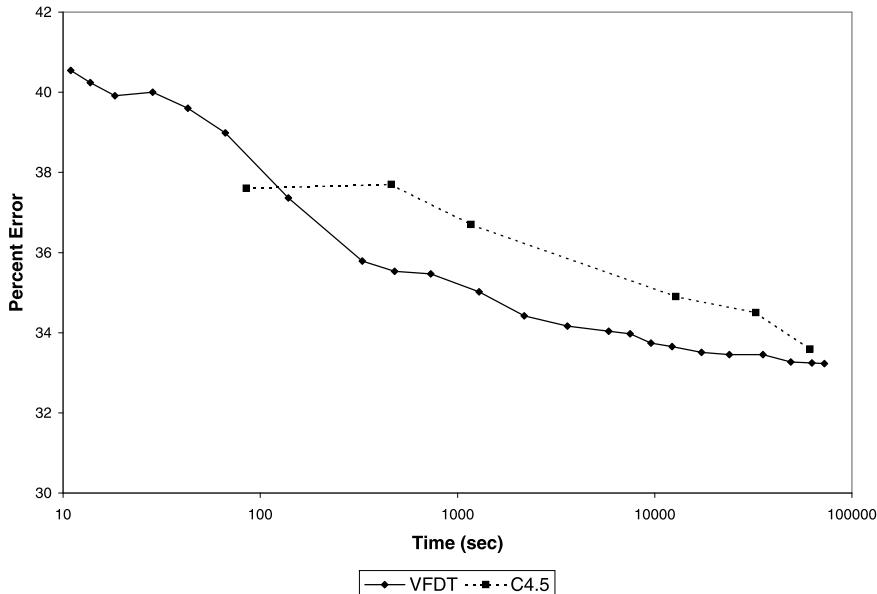


Fig. 10 Time vs. Error Rate on the ‘5 min’ data set

and accuracy advantage; they show the amount of learning time on a logarithmic scale on the x -axis and the error rate of the model learned in the indicated amount of time on the y -axis. VFDT has a more accurate model than C4.5 after nearly any amount of learning time (the region where C4.5 has an advantage in Fig. 8 is only about 1.6 % of the entire run). We attribute this to VFDT’s sampling method which allows it to use just the right amount of data to make each decision, while C4.5 must use all data for every decision.

5 Related Work

There has been a great deal of work on scaling up learning algorithms to very large data sets. Perhaps the most relevant falls into the general framework of sequential analysis [24]. Examples of relevant machine learning algorithms that use these ideas include PALO [10], Hoeffding races [16], the sequential induction model [9], AdaSelect [3], and the sequential sampling algorithm [20]. The core idea in these is that training samples should be accumulated (roughly) one at a time until some quality criteria is met, and no longer. Our work extends these in several ways, including providing stronger guarantees and finer-grained bounds, addressing memory issues, caching training examples, etc. Further, we have conducted extensive empirical tests of our framework on synthetic and real-world data sets, and we have applied it to a wide variety of learning algorithms.

Our method bears an interesting relationship to recent work in computational learning theory that uses algorithm-specific and run-time information to obtain better loss bounds (e.g., [6, 22]). A first key difference is that we attempt to bound a learner’s loss relative to the same learner running on infinite data, instead of relative to the best possible model (or the best possible from a given class). A second key difference is that we make more extensive use of run-time information in our bounds. These two changes make possible realistic bounds for widely-used learners, as exemplified in this chapter.

BOAT [8] is a scalable, incremental decision tree induction algorithm that uses a sampling strategy that allows it to learn several levels of a decision tree in a single pass over training data. Perhaps the most closely related system to our approach is the combination of BOAT with the DEMON framework [7]. DEMON was designed to help adapt any incremental learning algorithms to work effectively with data streams. DEMON assumes data arrives periodically in large blocks. Roughly, DEMON builds a model on every interesting suffix of the data blocks that are in a sliding window. For example, if the window contains n data blocks B_1, \dots, B_n , DEMON will have a model for blocks B_1, \dots, B_n ; a model for blocks B_2, \dots, B_n ; a model for B_3, \dots, B_n ; etc. When a new block of data (block B_{n+1}) arrives, DEMON incrementally refines the model for B_2, \dots, B_n to quickly produce the n block model for the new window. It then uses offline processing time to add B_{n+1} to all of the other models it is maintaining. DEMON’s advantage is its simplicity: it can easily be applied to any existing incremental learning algorithm. The disadvantages is the granularity at which it incorporates data (in periodic blocks compared to every example for VFDT) and the overhead required to store (and update) all of the required models.

6 Conclusion

In this chapter, we described VFDT, an incremental any-time decision tree induction algorithm capable of learning from massive data streams within stringent performance criteria. The core of the algorithm samples from the data stream exactly as much data as needed to make each induction decision. We evaluated VFDT with extensive studies on synthetic data sets, and with an application to a massive real-world data stream. We found that VFDT was able to outperform traditional systems by running faster and by taking advantage of additional data to learn higher quality models. A publicly available implementation of VFDT is available as part of the VFML library [13].

References

1. L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone, *Classification and Regression Trees* (Wadsworth, Belmont, 1984)

2. J. Catlett, Megainduction: machine learning on very large databases. PhD thesis, Basser Department of Computer Science, University of Sydney, Sydney, Australia (1991)
3. C. Domingo, R. Gavalda, O. Watanabe, Adaptive sampling methods for scaling up knowledge discovery algorithms. *Data Min. Knowl. Discov.* **6**, 131–152 (2002)
4. P. Domingos, G. Hulten, A general method for scaling up machine learning algorithms and its application to clustering, in *Proceedings of the Eighteenth International Conference on Machine Learning*, Williamstown, MA (Morgan Kaufmann, San Mateo, 2001), pp. 106–113
5. P. Domingos, G. Hulten, Learning from infinite data in finite time, in *Advances in Neural Information Processing Systems*, vol. 14, ed. by T.G. Dietterich, S. Becker, Z. Ghahramani (MIT Press, Cambridge, 2002), pp. 673–680
6. Y. Freund, Self bounding learning algorithms, in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, Madison, WI (Morgan Kaufmann, San Mateo, 1998)
7. V. Ganti, J. Gehrke, R. Ramakrishnan, DEMON: mining and monitoring evolving data, in *Proceedings of the Sixteenth International Conference on Data Engineering*, San Diego, CA (2000), pp. 439–448
8. J. Gehrke, V. Ganti, R. Ramakrishnan, W.-L. Loh, BOAT: optimistic decision tree construction, in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA (ACM, New York, 1999), pp. 169–180
9. J. Gratch, Sequential inductive learning, in *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR (AAAI Press, Menlo Park, 1996), pp. 779–786
10. R. Greiner, PALO: a probabilistic hill-climbing algorithm. *Artif. Intell.* **84**, 177–208 (1996)
11. W. Hoeffding, Probability inequalities for sums of bounded random variables. *J. Am. Stat. Assoc.* **58**, 13–30 (1963)
12. G. Hulten, P. Domingos, Mining complex models from arbitrarily large databases in constant time, in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Canada (ACM Press, New York, 2002), pp. 525–531
13. G. Hulten, P. Domingos, VFML—a toolkit for mining high-speed time-changing data streams (2003). <http://www.cs.washington.edu/dm/vfml/>
14. G. Hulten, P. Domingos, Y. Abe, Mining massive relational databases, in *IJCAI 2003 Workshop on Learning Statistical Models from Relational Data*, Acapulco, Mexico (2003)
15. G. Hulten, L. Spencer, P. Domingos, Mining time-changing data streams, in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, CA (ACM, New York, 2001), pp. 97–106
16. O. Maron, A. Moore, Hoeffding races: accelerating model selection search for classification and function approximation, in *Advances in Neural Information Processing Systems 6*, ed. by J.D. Cowan, G. Tesauro, J. Alspector (Morgan Kaufmann, San Mateo, 1994)
17. M. Mehta, A. Agrawal, J. Rissanen, SLIQ: a fast scalable classifier for data mining, in *Proceedings of the Fifth International Conference on Extending Database Technology*, Avignon, France (Springer, Berlin, 1996), pp. 18–32
18. R. Musick, J. Catlett, S. Russell, Decision theoretic subsampling for induction on large databases, in *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, MA (Morgan Kaufmann, San Mateo, 1993), pp. 212–219
19. J.R. Quinlan, *C4.5: Programs for Machine Learning* (Morgan Kaufmann, San Mateo, 1993)
20. T. Scheffer, S. Wrobel, Incremental maximization of non-instance-averaging utility functions with applications to knowledge discovery problems, in *Proceedings of the Eighteenth International Conference on Machine Learning*, Williamstown, MA (Morgan Kaufmann, San Mateo, 2001), pp. 481–488
21. J.C. Shafer, R. Agrawal, M. Mehta, SPRINT: a scalable parallel classifier for data mining, in *Proceedings of the Twenty-Second International Conference on Very Large Databases*, Bombay, India (Morgan Kaufmann, San Mateo, 1996), pp. 544–555
22. J. Shawe-Taylor, P.L. Bartlett, R.C. Williamson, M. Anthony, Structural risk minimization over data-dependent hierarchies. Technical report NC-TR-96-053, Department of Computer Science, Royal Holloway, University of London, Egham, UK (1996)

23. P.E. Utgoff, An improved algorithm for incremental induction of decision trees, in *Proceedings of the Eleventh International Conference on Machine Learning*, New Brunswick, NJ (Morgan Kaufmann, San Mateo, 1994), pp. 318–325
24. A. Wald, *Sequential Analysis* (Wiley, New York, 1947)
25. A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, H. Levy, Organization-based analysis of Web-object sharing and caching, in *Proceedings of the Second USENIX Conference on Internet Technologies and Systems*, Boulder, CO (1999), pp. 25–36

Frequent Itemset Mining over Data Streams

Gurmeet Singh Manku

1 Problem Definition

We study the problem of computing *frequent elements* in a data-stream. Given support threshold $s \in [0, 1]$, an element is said to be frequent if it occurs more than sN times, where N denotes the current length of the stream. If we maintain a list of counters of the form $\langle \text{element}, \text{count} \rangle$, one counter per unique element encountered, we need N counters in the worst-case. Many distributions are heavy-tailed in practice, so we would need far fewer than N counters. However, the number would still exceed $1/s$, which is the maximum possible number of frequent elements. If we insist on identifying *exact* frequency counts, then $\Omega(N)$ space is necessary. This observation motivates the definition of ϵ -approximate frequency counts : given support threshold $s \in [0, 1]$ and an error parameter $\epsilon \in (0, s)$, the goal is to produce a list of elements along with their estimated frequencies, such that three properties are satisfied:

- I. Estimated frequencies are less than the true frequencies by at most ϵN .
- II. All elements whose true frequency exceeds sN are output.
- III. No element whose true frequency is less than $(s - \epsilon)N$ is output.

Example Imagine a statistician who wishes to identify elements whose frequency is at least 0.1 % of the entire stream seen so far. Then the support threshold is $s = 0.1\%$. The statistician is free to set $\epsilon \in (0, s)$ to whatever she feels is a comfortable margin of error. Let us assume she chooses $\epsilon = 0.01\%$ (one-tenth of s). As per Property I, estimated frequencies are less than their true frequencies by at most 0.01 %. As per Property II, all elements with frequency exceeding $s = 0.1\%$ will be

G.S. Manku (✉)
Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA, USA
e-mail: manku@google.com

output; there are *no false negatives*. As per Property III, no element with frequency below 0.09 % will be output. This leaves elements with frequencies between 0.09 % and 0.1 %—these might or might not form part of the output. On the whole, the approximation has two aspects: high frequency false positives, and small errors in individual frequencies. Both kinds of errors are tolerable in real-world applications.

2 One-Pass Algorithms

We present three algorithms for computing ϵ -approximate frequency counts. To avoid floors and ceilings, we will assume that $1/\epsilon$ is an integer (if not, we can scale down ϵ to 2^{-r} where r is an integer satisfying $2^{-r} < \epsilon < 2^{-r+1}$). The data structure for all three algorithms is a list of counters of the form $\langle \text{element}, \text{count} \rangle$, initially empty. At any time, ϵ -approximate frequency counts can be retrieved by identifying those elements whose associated count exceeds $(s - \epsilon)N$. The algorithms differ in terms of the rules employed for creating, incrementing, decrementing and deleting counters.

MISRA–GRIES ALGORITHM ([7]). Let e denote a newly-arrived element. If a counter for e already exists, it is incremented. Otherwise, if there already exist $1/\epsilon$ counters, we *repeatedly diminish all counters* by 1 until some counter drops to zero. We then delete all counters with count zero, and create a new counter of the form $\langle e, 1 \rangle$.

LOSSY COUNTING ([6]). Let e denote a newly-arrived element. If a counter for e already exists, it is incremented. Otherwise, we create a new counter of the form $\langle e, 1 \rangle$. Whenever N , the current size of the stream, equals i/ϵ for some integer i , all counters are decrement by one—we discard any counter that drops to zero.

STICKY SAMPLING ([6]). The algorithm is randomized, with δ denoting the probability of failure. The algorithm maintains r , the sampling rate, which varies over the lifetime of the stream. Initially, $r = 1$. Let e denote the newly-arrived element. If a counter for e exists, it is incremented. Otherwise, we toss a coin with probability of success r . If the coin toss succeeds, we create an entry of the form $\langle e, 1 \rangle$; otherwise, we ignore e .

The sampling rate r varies as follows: Let $t = \frac{1}{\epsilon} \log(s^{-1}\delta^{-1})$. The first $2t$ elements are sampled at rate $r = 1$, the next $2t$ elements are sampled at rate $r = 1/2$, the next $4t$ elements are sampled at rate $r = 1/4$, and so on. Whenever the sampling rate changes, we update existing counters as follows: For each counter, we repeatedly toss an unbiased coin until the coin toss is successful, decrementing the counter for every unsuccessful outcome; if the counter drops to zero during this process, we delete the counter. Effectively, the new list of counters is identical to exactly the list that would have emerged, had we been sampling with the new rate from the very beginning.

Theorem 1 MISRA–GRIES ALGORITHM allows retrieval of ϵ -approximate frequency counts using at most $\frac{1}{\epsilon}$ counters.

Proof Consider a fixed element e . Whenever a counter corresponding to e is diminished by 1, $1/\epsilon - 1$ other counters are also diminished. Clearly, when N elements have been seen, a counter for e could not have been diminished by more than ϵN . \square

Theorem 2 LOSSY COUNTING allows retrieval of ϵ -approximate frequency counts using at most $\frac{1}{\epsilon} \log(\epsilon N)$ counters.

Proof Imagine splitting the stream into buckets of size $w = 1/\epsilon$ each. Let $N = Bw$, where B denotes the total number of buckets that we have seen. For each $i \in [1, B]$, let d_i denote the number of counters which were created when bucket $B - i + 1$ was active, i.e., the length of the stream was in the range $[(B - i)w + 1, (B - i + 1)w]$. The element corresponding to such a counter must occur at least i times in buckets $B - i + 1$ through B ; otherwise, the counter would have been deleted. Since the size of each bucket is w , we get the following constraints:

$$\sum_{i=1}^j id_i \leq jw \quad \text{for } j = 1, 2, \dots, B. \quad (1)$$

We prove the following set of inequalities by induction:

$$\sum_{i=1}^j d_i \leq \sum_{i=1}^j \frac{w}{i} \quad \text{for } j = 1, 2, \dots, B. \quad (2)$$

The base case ($j = 1$) follows from (1) directly. Assume that (2) is true for $j = 1, 2, \dots, p - 1$. We will show that it is true for $j = p$ as well. Adding $p - 1$ inequalities of type (2) (one inequality each for i varying from 1 to $p - 1$) to an inequality of type (1) (with $j = p$) yields

$$\begin{aligned} & \sum_{i=1}^p id_i + \sum_{i=1}^1 d_i + \sum_{i=1}^2 d_i + \dots + \sum_{i=1}^{p-1} d_i \\ & \leq pw + \sum_{i=1}^1 \frac{w}{i} + \sum_{i=1}^2 \frac{w}{i} + \dots + \sum_{i=1}^{p-1} \frac{w}{i}. \end{aligned}$$

Upon rearrangement, we get $p \sum_{i=1}^p d_i \leq pw + \sum_{i=1}^{p-1} \frac{(p-i)w}{i}$, which readily simplifies to (2) for $j = p$. This completes the induction step. The maximum number of counters is $\sum_{i=1}^B d_i \leq \sum_{i=1}^B \frac{w}{i} \leq \frac{1}{\epsilon} \log B = \frac{1}{\epsilon} \log(\epsilon N)$. \square

Theorem 3 STICKY SAMPLING computes ϵ -approximate frequency counts, with probability at least $1 - \delta$, using at most $\frac{2}{\epsilon} \log(s^{-1} \delta^{-1})$ counters in expectation.

Proof The expected number of counters is $2t = \frac{2}{\epsilon} \log(s^{-1} \delta^{-1})$. When $r \leq 1/2$, then $N = rt + rt'$, for some $t' \in [1, t]$. It follows that $\frac{1}{r} \geq \frac{t}{N}$. Any error in the estimated

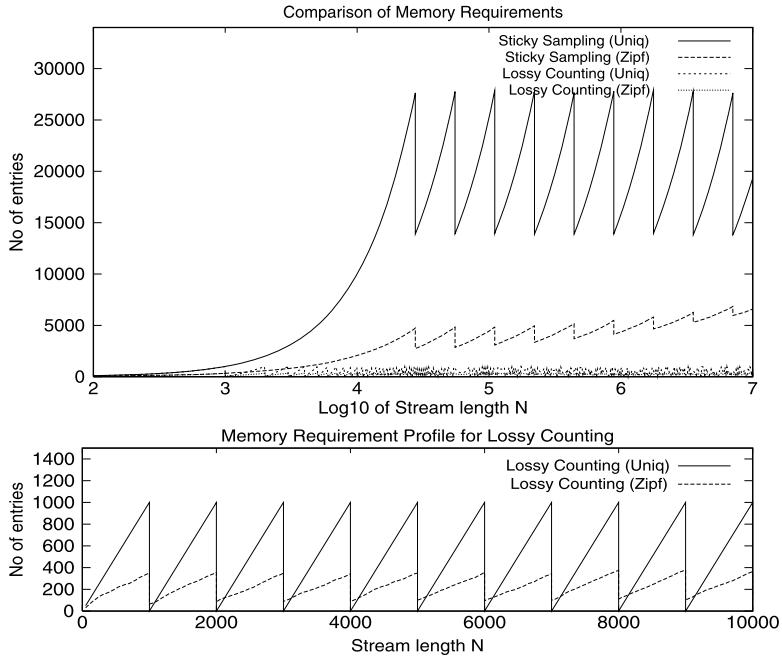


Fig. 1 Number of counters for support threshold $s = 1\%$, error parameter $\epsilon = 0.1\%$, and probability of failure $\delta = 10^{-4}$. Zipf denotes a Zipfian distribution with parameter 1.25. Uniq denotes a stream with no duplicates. The *bottom figure* magnifies a section of the barely-visible lines in the graph above

frequency of an element e corresponds to a sequence of unsuccessful coin tosses during the first few occurrences of e . The probability that this error exceeds ϵN is at most $(1 - \frac{1}{r})^{\epsilon N} \leq (1 - \frac{t}{N})^{-\epsilon N} \leq e^{-\epsilon t}$.

There are at most $1/s$ elements whose true frequency exceeds sN . The probability that the estimate frequency of *any* of them is deficient by ϵN , is at most $e^{-\epsilon t}/s$. Since $t \geq \frac{1}{\epsilon} \log(s^{-1}\delta^{-1})$, this probability is at most δ . \square

Each of the algorithms has its strengths that make it useful in different contexts. The MISRA–GRIES ALGORITHM has optimal worst-case space complexity and $O(1)$ amortized cost of update per element. The update cost has been improved to $O(1)$ worst-case by Karp et al. [5]. STICKY SAMPLING is useful for identifying ϵ -approximate frequent counts over sliding windows (see Arasu and Manku [2]). LOSSY COUNTING is useful when the input stream has duplicates and when the input distribution is heavy-tailed, as borne out by Fig. 1. The kinks in the curve for STICKY SAMPLING correspond to re-sampling. They are $\log_{10} 2$ units apart on the X-axis. The kinks for LOSSY COUNTING correspond to $N = i/\epsilon$ (when deletions occur). STICKY SAMPLING performs worse because of its tendency to remember every unique element that gets sampled. LOSSY COUNTING, on the other hand, is good at pruning low frequency elements quickly; only high frequency elements sur-

vive. For skewed distributions, both algorithms require much less space than their worst-case bounds in Theorems 2 and 3. LOSSY COUNTING is superior to MISRA–GRIES ALGORITHM for skewed data. For example, with $\epsilon = 0.01\%$, roughly 2000 entries suffice, which is only 20 % of $\frac{1}{\epsilon}$.

3 Frequent Itemset Mining

Let \mathcal{I} denote the universe of all *items*. Consider a stream of *transactions*, where each transaction is a subset of \mathcal{I} . An itemset $X \subseteq \mathcal{I}$ is said to have support s if X is a subset of at least sN transactions, where N denotes the length of the stream, i.e., the number of transactions seen so far. The frequent itemsets problem seeks to identify all itemsets whose support exceeds a user-specified *support threshold* s .

Frequent itemsets are useful for identifying *association rules* (see Agrawal and Srikant [1] for a seminal paper that popularized the problem). Considerable work in frequent itemsets has focused on devising data structures for compactly representing frequent itemsets, and showing how the data structure can be constructed in a few passes over a large disk-resident dataset. The best-known algorithm take two passes.

Identification of frequent itemsets over data streams is useful in a data-warehousing environment where bulk updates occur at regular intervals of time, e.g., daily, weekly or monthly. *Summary data structures* that store aggregates like frequent itemsets should be maintained *incrementally* because a complete rescan of the entire warehouse-database per bulk-update is prohibitively costly. The summary data structure should be significantly smaller than the warehouse-database but need not fit in main memory.

In a data stream scenario, where only one pass is possible, we relax the problem definition to compute ϵ -*approximate frequent itemsets*: Given support threshold $s \in (0, 1)$ and error parameter $\epsilon \in (0, s)$, the goal is to produce itemsets, along with their estimated frequencies, satisfying three properties:

- I. Estimated frequencies are less than the true frequencies by at most ϵN .
- II. All itemsets whose true frequency exceeds sN are output.
- III. No itemset whose true frequency is less than $(s - \epsilon)N$ is output.

We now develop an algorithm based upon LOSSY COUNTING for tackling the ϵ -*approximate frequent itemsets problem*. We begin by describing some modifications to LOSSY COUNTING.

3.1 Modifications to LOSSY COUNTING

We divide the stream into buckets of size $1/\epsilon$ each. Buckets are numbered sequentially, starting with 1. We maintain counters of the form $\langle \text{element}, \text{count}, \text{bucket_id} \rangle$, where bucket_id denotes the ID of the bucket that was active when this counter was

created. At bucket boundaries, we check whether $\text{count+bucket_id} \leq \epsilon N$. If so, the counter is deleted. This is equivalent to our earlier approach of decrementing counters at bucket boundaries and deleting those counters that drop to zero. The maximum possible error in the estimated frequency of an element is given by its `bucket_id` (which might be much less than ϵN). Furthermore, the algorithm continues to be correct even if we do not check the counters at each and every bucket boundary. However, the longer we defer the checks, the larger the space requirements due to the presence of noise (low-frequency elements in recent buckets).

3.2 Frequent Itemsets Algorithm

The input to the algorithm is a stream of transactions. The user specifies two parameters, support threshold, s , and error parameter, ϵ . We denote the current length of the stream by N . We maintain a data structure \mathcal{D} consisting of a set of entries of the form (set, f, Δ) , where set is an itemset (subset of \mathcal{I}), f is an integer representing the estimated frequency of set , and Δ is the maximum possible error in f . Initially, \mathcal{D} is empty.

The stream is divided into *buckets* consisting of $w = \lceil 1/\epsilon \rceil$ transactions each. Buckets are labeled with *bucket ids*, starting from 1. We denote the *current bucket id* by b_{current} . We do not process the stream transaction by transaction. Instead, we fill available main memory with as many transactions as possible, and then process the resulting *batch* of transactions together. Let β denote the number of buckets in main memory in the current batch being processed. We update \mathcal{D} as follows:

- **UPDATE_SET.** For each entry $(set, f, \Delta) \in \mathcal{D}$, update f by counting the occurrences of set in the current batch. If the updated entry satisfies $f + \Delta \leq b_{\text{current}}$, we delete this entry.
- **NEW_SET.** If a set set has frequency $f \geq \beta$ in the current batch and set does not occur in \mathcal{D} , we create a new entry $(set, f, b_{\text{current}} - \beta)$.

A set set whose true frequency $f_{set} \geq \epsilon N$, has an entry in \mathcal{D} . Also, if an entry $(set, f, \Delta) \in \mathcal{D}$, then the true frequency f_{set} satisfies the inequality $f \leq f_{set} \leq f + \Delta$. When a user requests a list of items with threshold s , we output those entries in \mathcal{D} where $f \geq (s - \epsilon)N$.

It is important that β be a large number. The reason is that any subset of \mathcal{I} that occurs $\beta + 1$ times or more, contributes an entry to \mathcal{D} . For small values of β , \mathcal{D} is polluted by noise (subsets whose overall frequency is very low, but which occur at least $\beta + 1$ times in the last β buckets).

Two design problems emerge: *What is an efficient representation of \mathcal{D} ? What is an efficient algorithm to implement UPDATE_SET and NEW_SET?*

3.3 Data Structure and Algorithm Design

We have three modules: TRIE, BUFFER, and SETGEN. TRIE is an efficient implementation of \mathcal{D} . BUFFER repeatedly reads in batches of transactions into available main memory and carries out some pre-processing. SETGEN then operates on the current batch of transactions in BUFFER. It enumerates subsets of these transactions along with their frequencies, limiting the enumeration using some pruning rules. Effectively, SETGEN implements the UPDATE_SET and NEW_SET operations to update TRIE. The challenge, it turns out, lies in designing a space-efficient TRIE and a time-efficient SETGEN.

TRIE. This module maintains the data structure \mathcal{D} outlined in Sect. 3.2. Conceptually, it is a forest (a set of trees) consisting of labeled nodes. Labels are of the form $\langle item_id, f, \Delta, level \rangle$, where $item_id$ is an item-id, f is its estimated frequency, Δ is the maximum possible error in f , and $level$ is the distance of this node from the root of the tree it belongs to. The root nodes have level 0. The level of any other node is one more than that of its parent. The children of any node are ordered by their item-id's. The root nodes in the forest are also ordered by item-id's. A node in the tree represents an itemset consisting of item-id's in that node and all its ancestors. There is a 1-to-1 mapping between entries in \mathcal{D} and nodes in TRIE.

To make the TRIE compact, we maintain *an array of entries* of the form $\langle item_id, f, \Delta, level \rangle$ corresponding to the pre-order traversal of the underlying trees. This is equivalent to a *lexicographic ordering of all the subsets encoded by the trees*. There are no pointers from any node to its children or its siblings. The *level*'s compactly encode the underlying tree structure. Such a representation suffices because tries are always scanned sequentially, as we show later.

Tries are used by several Association Rules algorithms, hash tries [1] being a popular choice. Popular implementations of tries require pointers and variable-sized memory segments (because the number of children of a node changes over time). Our TRIE is quite different.

BUFFER. This module repeatedly fills available main memory with a batch of transactions. Each transaction is a set of item-id's. Transactions are laid out one after the other in a big array. A bitmap is used to remember transaction boundaries. A bit per item-id denotes whether this item-id is the last member of some transaction or not. After reading in a batch, BUFFER sorts each transaction by its item-id's.

SETGEN. This module generates subsets of item-id's along with their frequencies in the current batch of transactions in lexicographic order. It is important that not all possible subsets be generated. A glance at the description of UPDATE_SET and NEW_SET operations reveals that a subset must be enumerated iff either it occurs in TRIE or its frequency in the current batch exceeds β . SETGEN uses the following pruning rule:

If a subset S does not make its way into TRIE after application of both UPDATE_SET and NEW_SET, then no supersets of S should be considered.

This is similar to the Apriori pruning rule [1]. We describe an efficient implementation of SETGEN in greater detail later.

Overall Algorithm

BUFFER repeatedly fills available main memory with a batch of transactions, and sorts them. SETGEN operates on BUFFER to generate sets of itemsets along with their frequency counts in lexicographic order. It limits the number of subsets using the pruning rule. Together, TRIE and SETGEN implement the UPDATE_SET and NEW_SET operations.

3.4 Efficient Implementations

In this section, we outline important design decisions that contribute to an efficient implementation.

BUFFER. If item-id's are successive integers from 1 thru $|\mathcal{I}|$, and if \mathcal{I} is small enough (say, less than 1 million), we maintain *exact frequency counts* for all items. For example, if $|\mathcal{I}| = 10^5$, an array of size 0.4 MB suffices. If exact frequency counts are available, BUFFER first prunes away those item-id's whose frequency is less than ϵN , and then sorts the transactions, where N is the length of the stream up to and including the current batch of transactions.

TRIE. As SETGEN generates its sequence of sets and associated frequencies, TRIE needs to be updated. Adding or deleting TRIE nodes *in situ* is made difficult by the fact that TRIE is a compact array. However, we take advantage of the fact that the sets produced by SETGEN (and therefore, the sequence of additions and deletions) are lexicographically ordered. Since our compact TRIE also stores its constituent subsets in their lexicographic order, the two modules: SETGEN and TRIE work hand in hand.

We maintain TRIE not as one huge array, but as a collection of fairly large-sized chunks of memory. Instead of modifying the original trie in place, we create a new TRIE afresh. Chunks belonging to the old TRIE are freed as soon as they are not required. Thus, the overhead of maintaining two TRIES is not significant. By the time SETGEN finishes, the chunks belonging to the old trie have been completely discarded.

For finite streams, an important TRIE optimization pertains to the last batch of transactions when the value of β , the number of buckets in BUFFER, could be small. Instead of applying the rules in Sect. 3.2, we prune nodes in the trie more aggressively by setting the threshold for deletion to sN instead of $b_{current} \approx \epsilon N$. This is because the lower frequency nodes do not contribute to the final output.

SETGEN. This module is the bottleneck in terms of time. Therefore, it merits careful design and run-time optimizations. SETGEN employs a priority queue called *Heap* which initially contains pointers to smallest item-id's of all transactions in BUFFER. Duplicate members (pointers pointing to the same item-id) are maintained together and they constitute a single entry in *Heap*. In fact, we chain all the pointers together, deriving the space for this chain from BUFFER itself. When an item-id in BUFFER is inserted into *Heap*, the 4-byte integer used to represent an item-id is converted into a 4-byte pointer. When a heap entry is removed, the pointers are restored back to item-id's.

SETGEN repeatedly processes the smallest item-id in *Heap* to generate singleton sets. If this singleton belongs to TRIE after UPDATE_SET and NEW_SET rules have been applied, we try to generate the next set in lexicographic sequence by extending the current singleton set. This is done by invoking SETGEN recursively with a new heap created out of successors of the pointers to item-id's just removed and processed. The successors of an item-id is the item-id following it in its transaction. Last item-id's of transactions have no successors. When the recursive call returns, the smallest entry in *Heap* is removed and all successors of the currently smallest item-id are added to *Heap* by following the chain of pointers described earlier.

3.5 System Issues and Optimizations

BUFFER scans the incoming stream by memory mapping the input file. This saves time by getting rid of double copying of file blocks. The UNIX system call for memory mapping files is `mmap()`. The accompanying `madvise()` interface allows a process to inform the operating systems of its intent to read the file sequentially. We used the standard `qsort()` to sort transactions. The time taken to read and sort transactions pales in comparison with the time taken by SETGEN, obviating the need for a custom sort routine. Threading SETGEN and BUFFER would not help because SETGEN is significantly slower.

Tries are written and read sequentially. They are operational when BUFFER is being processed by SETGEN. At this time, the disk is idle. Further, the rate at which tries are scanned (read/written) is much smaller than the rate at which sequential disk I/O can be done. It is indeed possible to maintain TRIE on disk without any loss in performance. This has two important advantages:

- (a) The size of a trie is not limited by the size of main memory available. This means that the algorithm can function even when the amount of main memory available is quite small.
- (b) Since most available memory can be devoted to BUFFER, we can work with tiny values of ϵ . This is a big win.

Memory requirements for *Heap* are modest. Available main memory is consumed primarily by BUFFER, assuming TRIES are on disk. Our implementation allows the user to specify the size of BUFFER.

On the whole, the algorithm has two unique features: there is *no candidate generation phase*, which is typical of Apriori-style algorithms. Further, the idea of using compact disk-based tries is novel. It allows us to compute frequent itemsets under low memory conditions. It also enables our algorithm to handle smaller values of support threshold than previously possible.

Experimental evaluation over a variety of datasets is available in [6].

4 Applications and Related Work

Frequency counts and frequent itemsets arise in a variety of applications. We describe two of these below.

4.1 Iceberg Queries

The idea behind Iceberg Queries[4] is to identify aggregates in a GROUP BY of a SQL query that exceed a user-specified threshold τ . A prototypical query on a relation $R(c_1, c_2, \dots, c_k, \text{rest})$ with threshold τ is

```
SELECT      c1, c2, ..., ck, COUNT(rest)
FROM        R
GROUP BY    c1, c2, ..., ck
HAVING     COUNT(rest) >=  $\tau$ 
```

The parameter τ is equivalent to $s|R|$ where s is a percentage and $|R|$ is the size of R . The frequent itemset algorithm developed in Sect. 3 runs in only one pass, and out-performs the highly-tuned algorithm in [4] that uses repeated hashing over multiple passes.

4.2 Network Flow Identification

Measurement and monitoring of network traffic is required for management of complex Internet backbones. In this context, identifying flows in network traffic is an important problem. A flow is defined as a sequence of transport layer (TCP/UDP) packets that share the same source+destination addresses. Estan and Verghese [3] recently proposed algorithms for identifying flows that exceed a certain threshold, say 1 %. Their algorithms are a combination of repeated hashing and sampling, similar to those by Fang et al. [4] for Iceberg Queries.

4.3 Algorithms for Sliding Windows

Algorithms for computing approximate frequency counts over sliding windows have been developed by Arasu and Manku [2]. In a *fixed-size* sliding window, the size of the window remains unchanged. In a *variable-sized* sliding window, at each time-step, an adversary can either insert a new element, or delete the oldest element in the window. When the size of the window is W , the space-bounds for a randomized algorithm (based upon STICKY SAMPLING) are $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon\delta})$ and $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon\delta} \log \epsilon W)$ for fixed-size and variable-size windows respectively. The corresponding bounds for a deterministic algorithm (based upon MISRA–GRIES ALGORITHM) are $O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon})$ and $O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon} \log \epsilon W)$, respectively. It would be interesting to see if any of these algorithms can be adapted to compute ϵ -approximate frequent itemsets in a data stream.

References

1. R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in *Proc. of 20th Intl. Conf. on Very Large Data Bases* (1994), pp. 487–499
2. A. Arasu, G.S. Manku, Approximate counts and quantiles over sliding windows, in *Proc. ACM Symposium on Principles of Database Systems* (2004)
3. C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.* **21**(3), 270–313 (2003)
4. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. Ullman, Computing iceberg queries efficiently, in *Proc. of 24th Intl. Conf. on Very Large Data Bases* (1998), pp. 299–310
5. R.M. Karp, C.H. Papadimitriou, S. Shenker, A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.* **28**, 51–55 (2003)
6. G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in *Proc. 28th VLDB* (2002), pp. 356–357
7. J. Misra, D. Gries, Finding repeated elements. *Sci. Comput. Program.* **2**(2), 143–152 (1982)

Temporal Dynamics of On-Line Information Streams

Jon Kleinberg

1 Introduction

A number of recent computing applications involve information arriving continuously over time in the form of a *data stream*, and this has led to new ways of thinking about traditional problems in a variety of areas. In some cases, the rate and overall volume of data in the stream may be so great that it cannot all be stored for processing, and this leads to new requirements for efficiency and scalability. In other cases, the quantities of information may still be manageable, but the data stream perspective takes what has generally been a static view of a problem and adds a strong temporal dimension to it.

Our focus here is on some of the challenges that this latter issue raises in the settings of text mining, on-line information, and information retrieval. Many information sources have a stream-like structure, in which the way content arrives over time carries an essential part of its meaning. News coverage is a basic example; understanding the pattern of a developing news story requires considering not just the content of the relevant articles but also how they evolve over time. Some of the other basic corpora of interest in information retrieval—for example, scientific papers and patents—show similar temporal evolution over time-scales that can last years and decades. And the proliferation of on-line information sources and on-line forms of communication has led to numerous other examples: e-mail, chat, discussion boards, and weblogs (or “blogs”) all represent personal information streams with intricate topic modulations over time.

This survey was written in 2004 and circulated on-line as a preprint prior to its appearance in this volume.

J. Kleinberg (✉)

Department of Computer Science, Cornell University, 4105B Upson Hall, Ithaca, NY 14853, USA
e-mail: kleinber@cs.cornell.edu

Indeed, all these information sources co-exist on-line—news, e-mail, discussion, commentary, and the collective output of professional and research communities; they form much of the raw material through which Internet users navigate and search. They have also served to make the “time axis” of information increasingly visible. One could argue that these developments have led to a shift in our working metaphor for Internet and Web information, from a relatively static one, a “universal encyclopedia,” to a much more dynamic one, a current awareness medium characterized by the complex development of topics over time-scales ranging from minutes to years.

What are the right techniques for dealing with the temporal dynamics of information streams? Some of the complexities inherent in such streams can be seen in the paradigmatic examples of news and e-mail. Each provides a reader with a sequence of documents exhibiting a *braided* and *episodic* character: *braided*, in the sense that many parallel, topically coherent streams are merged and interwoven into the single stream that the reader sees; and *episodic*, in the sense that topics generally grow in intensity over a temporally coherent period, and then fade again. Despite these twin sources of complexity, however, these information streams in their raw forms lack any explicit organizing structure beyond the granularity of individual articles or messages. Thus, a first step in working with such information streams is to define appropriate structures that abstract their intertwined topics, their multiple bursty episodes, and their long-term trends.

In this survey we discuss a number of approaches that have been proposed in recent years for working with the temporal properties of information streams. In Sect. 2, we give an overview of some of the basic techniques; we begin with one of the earliest systematic efforts on these problems, the Topic Detection and Tracking initiative, and then go on to discuss three subsequent approaches—threshold-based methods, state-based methods, and trend-based methods—that search for different types of patterns in information streams. In Sect. 3, we discuss some recent applications of these techniques to the analysis of weblogs, queries to Web search engines, and usage data at high-traffic Web sites. Finally, we conclude in Sect. 4 with some thoughts on directions for further research.

Analyzing the temporal properties of these types of information streams is part of the broader area of *sequential pattern mining* within the field of data mining, and can be viewed as an application of *time-series analysis*, a fundamental area in statistics. In order to keep the scope of this survey manageable, we have not attempted to cover these more general topics; we refer the reader to the papers of Agrawal and Srikant and of Mannila et al. [2, 28] for some of the foundational work on sequential pattern mining, and to the text by Hand et al. [18] for further results in this area.

2 Techniques: Thresholds, State Transitions, and Trends

Topic Detection and Tracking

The Topic Detection and Tracking (TDT) research initiative represented perhaps the first systematic effort to deal with the issues raised by text information streams.

Observing that it is tricky to pin down the underlying set of problems that need to be solved, it sought to define a concrete set of tasks based on the general problem of identifying coherent topics in a stream of news stories.

Allan et al. [5] and Papka [30] describe some of the initial considerations that went into the formulation of these tasks. To begin with, the TDT researchers distinguished between the notion of a *topic*—a staple of information retrieval research—and an *event*, the latter referring to a unique occurrence that happened at a specific time. For example, “baseball” would be considered a topic, whereas “Game 6 of the 1975 World Series” would be considered an event. This kind of distinction is natural in the context of news coverage, although of course the conceptual boundary between topics and events is quite flexible. Further refinements of these definitions added other notions, such as *activities*.

Within this framework, one imagines a TDT system operating roughly as follows. Given a stream of news articles, the system should automatically recognize those stories that discuss an event it has not seen before, and should begin tracking each of these events so as to identify the sub-stream of further stories that discuss it. Implicit in this description is a pair of basic tasks that can be evaluated separately: *new event detection*, in which the goal is to recognize the first story to discuss an event, and *event tracking*, in which the goal is to group stories that discuss the same event. Other TDT tasks have been defined as well, including the *segmentation* of a continuous stream of news text into distinct stories; this is a necessary prerequisite for many of the other tasks, and crucial for transcriptions of audio broadcasts, where the boundaries between stories are not generally made explicit in a uniform way. Finally, a distinction is made between the *retrospective* versions of these tasks, in which the full corpus of news stories is available for analysis, and the *on-line* versions, in which decisions must be made in real-time as news stories appear. One challenge inherent in the detection and tracking tasks is that events can have very different temporal “shapes”: an unexpected event like a natural disaster comes on with a sharp spike, while an expected event like an election has a slow build-up and potentially a quicker decay after the event is over.

A range of different techniques from information retrieval have been shown to be effective on the TDT tasks, including clustering methods for event tracking that trade off between content similarity and temporal locality. The volume edited by Allan [4] covers much of the state of the art in this area. The general framework developed for the TDT project has proved useful in a number of subsequent efforts that have not directly used the TDT task descriptions. For example, recent work on the *Newsjunkie* system [15] sought techniques for determining the *novelty* in individual news stories, relative to previously seen stories concerned with the same general topic or event; quantifying novelty in this sense can be viewed as a relaxation of the task of new event detection.

Information Visualization

At roughly the same time as the TDT project, the information visualization community also began investigating some of the issues inherent in text information streams

[19, 29, 37]. The goal was to find visual metaphors by which users could navigate large collections of documents with an explicit temporal dimension.

The ThemeRiver system [19] depicts a text collection that evolves over time, such as a corpus of news stories, using a “river” metaphor: differently colored currents in the river indicate different topics in the collection, and the currents vary in width to indicate news stories that have a large volume of relevant articles at a particular point in time. Note that this visualization approach nicely captures the view, discussed above, of text information streams as a braided, episodic medium: the individual “braids” appear in ThemeRiver as the colored currents, while the episodes stand out as gradual or sudden widenings of a particular current.

Timelines and Threshold-Based Methods

Timelines are a common means of representing temporal data, appearing in computational applications (see, e.g., [6, 31]) and familiar from more traditional print media as well. Swan, Allan, and Jensen [33–35] considered the problem of creating timelines for document streams, again focusing on collections of news articles. They framed the problem as a selection task: the construction of a timeline should involve retrospectively identifying the small set of most significant episodes from a large stream of documents, associating a descriptive tag with each one, and displaying these tagged episodes in their natural temporal order. Formalizing this notion requires an implementable definition of *episodes* in a text stream, together with a way of ranking episodes by significance.

Swan et al. base their definition of episodes on time-varying *features* in the text—words, named entities, and noun features are all examples of possible features for this purpose. Now, each feature has an average rate at which it appears in the corpus; for a news stream, this would be the total number of occurrences of the feature divided by the number of days represented in the corpus. An episode is then associated with a contiguous interval of time during which one of these features exceeds its average rate by a specific threshold; Swan et al. determine this threshold on a per-feature basis using a χ^2 test, and group together consecutive days over which a given feature exceeds its threshold. Thus, for example, if we were following news articles early in a US Presidential election year, we might find that for a number of days in a row, the word “Iowa” appears a significant factor more frequently than it standardly does in the news, reflecting coverage of the Iowa caucuses. The most significant episodes computed this way can then be included in the timeline, each with a start and end time, and each associated with a specific word or phrase.

State-Based Methods

The number of occurrences of a given feature can be quite noisy, varying widely from one day to the next even in the middle of an event in which the feature figures prominently. Swan et al. observe that this poses difficulties in the construction of

intervals to represent episodes on a timeline; a feature being tracked may oscillate above and below the threshold, turning something that intuitively seems like a single long interval into a sequence of shorter ones, interrupted by the below-threshold days. Thus, to continue our example from above, there may not be a single interval associated with the word “Iowa” early in election coverage, but rather a set of short ones separated by small gaps. This was also an observation in the development of ThemeRiver, that Sundays and other days with lighter news coverage show up as recurring “pinch points” in the representation of the stream. Swan and Allan proposed heuristics to deal with this problem [34]; for example, one can merge two intervals corresponding to the same feature if they are separated by only a single day on which the feature was below threshold.

In [23], the author proposed a method for defining episodes in a stream of documents that deals with the underlying noise by explicitly positing a source model for the words in the documents. The motivation comes from queueing theory, where a bursty source of network traffic is often modeled as a probabilistic automaton: at any given point in time, the automaton can be in one of several different states, the rate at which traffic is generated is determined by this state, and transitions between states are determined probabilistically [7, 13, 21].

In the case of documents, one can imagine each word to be a type of “traffic” generated by its own source, modeled as an automaton. Some words are highly “bursty” in the sense that their frequency spikes when a particular event is in the news; in the probabilistic model, this corresponds to a contiguous interval of time during which the automaton is in a state corresponding to a high rate of traffic. Such *word bursts* are the intervals that correspond to discrete episodes in this model; one can identify them using a Viterbi-style algorithm [32], taking the times at which a given word occurs as input and computing the most likely sequence of states of the automaton that is assumed to be generating it. Note the distinction with the threshold-based methods described earlier: rather than associating episodes with intervals when the *observed rate* of the word is consistently high, it associates them with intervals in which the *state* of the underlying automaton has a high rate. Thus, by modeling state transitions as relatively low-probability events, the bursts that are computed by the algorithm tend to persist through periods of noise: remaining in a single state through such a noisy period is viewed as more probable than performing many transitions.

Moreover, an automaton with a large set of states corresponding to increasingly rapid rates can expose the natural nested structure of word bursts: in the midst of an elevated use of a particular term, the rate of use may increase further, producing a “burst within a burst.” This nesting can in principle be iterated arbitrarily, yielding a natural hierarchy. To continue our example of US presidential election coverage in the news, one could imagine over a multi-year period seeing bursts for the word “convention” in the summers of presidential election years, with two inner, stronger bursts corresponding to the Democratic and Republican conventions.

A timeline for a document stream can be naturally constructed using a two-state automaton associated with each word—a slower *base state* corresponds to the average rate of appearance of the word, while a *burst state* corresponds to a faster “burst

rate.” Using just two states does not expose any nested or hierarchical structure, but it makes it very simple to interpret the burst intervals. For each word, one computes the intervals during which the automaton is in the burst state. Now, by definition, in each such interval the automaton is more likely to be in the burst state than the base state; we define the *weight* of the interval to be the factor by which the probability of the burst state exceeds the probability of the base state over the course of the interval. Essentially, the weight thus represents our “confidence” that the automaton is indeed in the burst state. A timeline with a given number of items can simply be defined to consist of the intervals with the highest weight.

This approach is considered in the context of several different kinds of document streams in [23], including e-mail and titles of scientific papers. To give a sense for the kind of timelines that can be constructed this way, Table 1 shows an example from [23], the results of this method applied to the document stream consisting of titles of all papers from the database conferences SIGMOD and VLDB, 1975–2001. A two-state automaton is associated with each word, in which the burst state has twice the rate of the base state, and the 30 burst intervals of highest weight are depicted. Note that all words are included in the analysis, but, matching our intuition, we see that stop-words do not occur on the list because they do not tend to be very bursty. On the other hand, certain words appearing on the list do reflect trends in language use rather than in technical content; for example, the bursts for ‘data,’ ‘base,’ and ‘bases’ in the years 1975–1981 in Table 1 arise in large part from the fact that the term ‘database’ was written as two words in a significant number of the paper titles during this period.

Table 1 is primarily just an illustrative example; Mane and Börner have used this burst detection approach in a similar but much more extended way as part of a large-scale scientometric analysis of topics in the *Proceedings of the National Academy of Sciences* over the years 1982–2001 [27]. The bursty topics can be viewed as providing a small set of natural “entry points” into a much larger collection of documents; for example, they provide natural starting points for issuing queries to the collection, as well as the raw material for a clustering analysis of the content. As an instance of the latter application, Mane and Börner computed a two-dimensional representation of term co-occurrences for a collection of words and phrases selected to maximize a combination of burstiness and frequency, and this representation was then evaluated by domain experts.

Another observation based on the example in Table 1 is that the state-based model has the effect of producing longer bursts than a corresponding use of thresholds; although the burst state had twice the rate of the base state, most of the terms did not maintain a rate of twice their overall average throughout the entire interval. Indeed, only five terms in these paper titles appeared continuously at twice their average rate for a period of more than three years. David Jensen makes the observation that this kind of state-based smoothing effect may be more crucial for some kinds of text streams than for others: using just the titles of papers introduces a lot of noise that needs to be dealt with, while news articles tend to be written so that even readers joining the coverage many days into the event will still be able to follow the context [20].

Table 1 The 30 bursts of highest weight using titles of all papers from the database conferences SIGMOD and VLDB, 1975–2001

Word	Interval of burst
data	1975 SIGMOD—1979 SIGMOD
base	1975 SIGMOD—1981 VLDB
application	1975 SIGMOD—1982 SIGMOD
bases	1975 SIGMOD—1982 VLDB
design	1975 SIGMOD—1985 VLDB
relational	1975 SIGMOD—1989 VLDB
model	1975 SIGMOD—1992 VLDB
large	1975 VLDB—1977 VLDB
schema	1975 VLDB—1980 VLDB
theory	1977 VLDB—1984 SIGMOD
distributed	1977 VLDB—1985 SIGMOD
data	1980 VLDB—1981 VLDB
statistical	1981 VLDB—1984 VLDB
database	1982 SIGMOD—1987 VLDB
nested	1984 VLDB—1991 VLDB
deductive	1985 VLDB—1994 VLDB
transaction	1987 SIGMOD—1992 SIGMOD
objects	1987 VLDB—1992 SIGMOD
object-oriented	1987 SIGMOD—1994 VLDB
parallel	1989 VLDB—1996 VLDB
object	1990 SIGMOD—1996 VLDB
mining	1995 VLDB—
server	1996 SIGMOD—2000 VLDB
sql	1996 VLDB—2000 VLDB
warehouse	1996 VLDB—
similarity	1997 SIGMOD—
approximate	1997 VLDB—
web	1998 SIGMOD—
indexing	1999 SIGMOD—
xml	1999 VLDB—

In a sense, one can view a state-based approach as defining a more general, “relaxed” notion of a threshold; the optimization criterion inherent in determining a maximum-likelihood state sequence is implicitly determining how far above the base rate, and for how long, the rate must be (in an amortized sense) in order for the automaton to enter the burst state. Zhu and Shasha considered a more direct way of generalizing the threshold approach [38]. For each possible length k , they allow a user-defined threshold $t(k)$; any interval of length k containing at least $t(k)$ occurrences of the feature is declared to be a burst. Note that this means that a single

feature can have overlapping burst intervals of different lengths. They then go on to develop fast algorithms for enumerating all the bursts associated with a particular feature.

Trend-Based Methods

Thus far we have been discussing models for episodes as regions of increased density—words fluctuate in their patterns of occurrence, and we have been seeking short intervals in which a word occurs an unusual number of times. Liebscher and Belew propose a different structure of interest in a stream of documents: the sets of words that exhibit the most pronounced rising and falling trends over the entire length of the corpus [26]. For streams that are grouped over time into relatively few bins—such as yearly tabulations of research papers like we considered in Table 1—they apply linear regression to the set of frequencies of each term (first averaging each value with its two neighbors). They also argue that less sparse data would be amenable to more complex time-series analysis. Features with the most positive slopes represent the strongest rising terms, while those with the most negative slopes represent the strongest falling terms.

In addition to summarizing long-range trends in the stream, Liebscher and Belew also propose an application to *temporal term weighting*: if a user issues a query for a particular term that rises significantly over time, early documents containing the term can be weighted more heavily. The argument is that such early documents may capture more fundamental aspects of a topic that later grew substantially in popularity; without temporal weighting, it would be hard to favor such documents in the results of a search. A similar approach can be applied to terms that decrease significantly over time.

Two-Point Trends

A number of search engines have recently begun using the notion of rising and falling terms to present snapshots of trends in search behavior; see, for example, Google’s *Zeitgeist* and Ask’s *Top Searches* [8, 16]. The canonical approach for doing this is to compare the frequency of each search term in a given week to its frequencies in the previous week, and to find those for which the change has been largest.

Of course, this definition depends critically on how we choose to measure change. The tricky point here is that there are many natural ways to try quantifying a trend involving just two data points—i.e., in the case of the search engine application, the two data points are the frequencies of occurrence in the previous week and the current week. To make this concrete, suppose we have text from two different time periods, and we want to define the amount of *change* experienced by a given word w . We define each occurrence of each word to be a *token*, and let n_0 and n_1 denote the total number of token in the text from the first and second periods,

respectively. Let $f_0(w)$ and $f_1(w)$ denote the total number of occurrences of the word w in the first and second periods, respectively, and define $p_0(w) = f_0(w)/n_0$ and $p_1(w) = f_1(w)/n_1$.

Now, two basic ways to rank words by the significance of their “falling” and “rising” patterns would be to measure absolute change or relative change. We define the *absolute change* of w to be $f_1(w) - f_0(w)$; we can also define the *normalized absolute change* as $f_1(w)(n_0/n_1) - f_0(w)$ to correct for the fact that the amount of text from the first and second periods may differ significantly. Analogously, we define the *relative change* of w to be $f_1(w)/f_0(w)$ and the *normalized relative change* to be $(n_0 f_1(w))/(n_1 f_0(w))$. Charikar et al. [10], motivated by the application to trend detection at Google, discuss streaming algorithms for computing these types of change measures on large datasets. While the exact methods for measuring change in search frequency have not been made public, Ask Jeeves indicates that it ranks terms by a variant of relative change, while Charikar et al. suggest that Google ranks terms by a variant of absolute change.

If one thinks about it intuitively, absolute and relative change are each quite extreme measures for this task, though in opposite directions. In order to be among the most highly ranked for absolute change, a word must have a very high rate of occurrence; this means that the lists of most significant risers and fallers will be dominated by very common words, and even fluctuations in stop-word use can potentially swamp more interesting trends. On the other hand, in order to be among the most highly ranked for relative change, a word generally must occur extremely rarely in one of the two periods and have a massive spike in the other; such trends can often be artifacts of a particular transient pattern of usage. (As a degenerate case of this latter issue, it is not clear how to deal with words that failed to occur in one of the two periods.)

As an illustration of these issues, Table 2 shows the five most significant falling and rising words from the text of weekly US Presidential radio addresses, with the two periods consisting of the years 2002 and 2003. A ranking by normalized absolute change mainly puts stop-words in the top few positions, for the reason suggested above; mixed in with this are more topic-specific words that experienced huge fluctuations, like “Iraq.” The ranking by relative change is harder to interpret, but consists primarily of words that occurred very rarely in one year or the other.

Is there a principled way to interpolate between these two extremes, favoring topic-specific words that experienced large changes and that, while relatively frequent, were still not among the most overwhelmingly common words in the corpus? Our proposal here is that the state-based models considered earlier provide a very simple measure of change that performs such an interpolation. Recall that, in ranking bursts produced by a two-state automaton, we used a *weight* function that measured the probability of the burst relative to the probability of the automaton having remained in its base state. For the present application, we can imagine performing the following analogous calculation. Suppose that, after observing the first period, we posit the following model for generating the words in the second period: n_1 tokens will be generated independently at random, with the j th token taking the value w with probability $p_0(w)$. In other words, the observed word frequencies in

Table 2 Most prominent falling and rising words in text of weekly US Presidential radio addresses, comparing 2002 to 2003

Normalized absolute change:	
Falling words	Rising words
to	iraq
i	are
president	and
security	iraqi
it	of

Relative change:	
Falling words	Rising words
welfare	aids
she	rising
mexico	instead
created	showing
love	government

Probabilistic generative model:	
Falling words	Rising words
homeland	iraq
trade	iraqi
security	aids
senate	seniors
president	coalition

the first period are assumed to generate the tokens in the second period via a simple probabilistic generative model.

Now, the significance of an increase in word w in the second period is based simply on the probability that $f_1(w)$ tokens out of n_1 would take the value w ; given the model just defined, this is equal to

$$\binom{n_1}{f_1(w)} p_0(w)^{f_1(w)} (1 - p_0(w))^{n_1 - f_1(w)}.$$

(Since this will typically be a very small quantity, we work with logarithms to perform the actual calculation.) By choosing the words with $p_1(w) > p_0(w)$ for which this probability is lowest, we obtain a list of the top rising terms; the analogous computation produces a list of the top falling terms. The third part of Table 2 shows the results of this computation on the same set of US Presidential radio addresses; while we see some overlap with the previous two lists, the probabilistic generative model arguably identifies words in a way that is largely free from the artifacts introduced by the two simpler measures.

Of course, there are many other ways to interpolate between absolute and relative change. The point is simply that a natural generative model can produce results that are much cleaner than these cruder measures, and hence, without significantly

increasing the difficulty of the ranking computation, yield trend summaries of this type that are potentially more refined.

3 Applications: Weblogs, Queries, and Usage Data

Weblogs

Personal home pages have been ubiquitous on the Web since its initial appearance; they are the original example of spontaneous Web content creation on a large scale by users who in many cases are not technically sophisticated. *Weblogs* (also referred to as *blogs*) can be viewed as a more recent step in the evolution of this style of personal publishing, consisting generally of dated, journal-style entries with links and commentary. Whereas a typical home page may be relatively static, a defining feature of weblogs is this explicit temporal annotation and regular updating. A significant sub-population of webloggers focus on news events, commenting on items of interest that they feel are being ignored or misportrayed by traditional news organizations; in some cases, these weblogs have readerships of substantial size. As a result, it is also natural to consider the space of weblogs as containing a stream of news that parallels the mainstream news media, much more heterogeneous both in focus and in authoring style.

All this makes it natural to apply the type of temporal analysis we have been discussing to the information stream consisting of indexable weblog content. In one of the first such applications to this domain, Dan Chan, motivated in part by the word burst model of [23], implemented a word-burst tracker on his widely-used weblog search site Daypop [11]. As we saw earlier with bursty topics from collections of research papers [27], the Daypop application provides another basic example of the way in which a list of bursty words can provide natural entry points into a corpus—in this case to search for items of interest in current news and commentary. By including word bursts computed both from weblog text and from headlines in the mainstream news, the site allows one to identify both overlaps and differences in the emphases of these two parallel media.

In a roughly concurrent study, Kumar et al. focused on a combined analysis of weblog content and link structure [24]. Because so much of weblog discourse takes place through reference to what other webloggers are reading and writing, the temporal dynamics of the link structure conveys important information about emerging topics and discussion. Kumar et al. work on the problem of identifying *subgraph bursts* in this structure, consisting of a significant number of link appearances among a small set of sites in a short period of time. They observe that this poses greater computational challenges than the analogous search for bursty words: whereas word bursts can be identified one word at a time, subgraph bursts need a more global analysis, since they are not the result of activity at any one site in isolation.

From a methodological point of view, one appealing feature of weblogs as a domain is the richness of the data; in principle, one can track the growth of a topic at an

extremely fine-grained level, using both the detailed temporal annotations and the links that webloggers use to indicate where they learned a piece of information. Adar et al. [1] and Gruhl et al. [17] exploit this to determine not just *whether* a given topic burst has occurred, but to some extent *how* and *why* it occurred. This involves modeling the spread of the topic as a kind of epidemic on the link structure—identifying the “seeds” of the topic, where it appeared earliest in time, and then tracing out the process of *contagion* by which the topic spread from one weblog to another. Adar et al. use this to define a ranking measure for weblogs, based on an estimate of their ability to start such an epidemic process. Gruhl et al. attempt to learn parameters of these epidemic processes, using an instance of the *General Cascade Model* for the diffusion of information in social networks, proposed by Kempe et al. [22].

Search Engine Queries

The logs of queries made to a large search engine provide a rich domain for temporal analysis; here the text stream consists not of documents but of millions of very short query strings issued by search engine users. Earlier, we discussed a simple use of query logs to identify the most prominently rising and falling query terms from one week to the next. We now discuss two recent pieces of work that perform temporal analysis of such logs for the purpose of enhancing and improving search applications.

Vlachos et al. [36] study query logs from the MSN search engine, providing techniques to identify different types of temporal patterns in them. For example, the frequency of the query “cinema” has a peak every weekend, while the frequency of the query “Easter” build to a single peak each spring and then drops abruptly. Vlachos et al. apply a threshold-based technique to a moving average of the daily frequencies for a given query in order to find the burst periods for the query, and they propose a “query-by-burst” technique that can identify queries with burst periods that closely overlap in time. Using Fourier analysis, they also build a representation of the temporal periodicities in a query’s rate over time, and then apply time-series matching techniques to identify other queries with very similar temporal patterns—such similarity can be taken as a form of evidence in the identification of related queries.

Diaz and Jones [12] also build temporal profiles, but they use the timestamps of the documents *returned* in response to a query, rather than the timestamps of the invocations of the query by users. Thus, in a corpus of news articles, a query about a specific natural disaster will tend to produce articles that are tightly grouped around a single date. Such temporal profiles become features of a query that can be integrated into a probabilistic model of document relevance; experiments in [12] show that the use of such features can lead to performance improvements.

Usage Data

On-line activity involves user behavior over short time-scales—browsing, searching, and communicating—as well as the creation of less ephemeral written content,

including the text of pages and files on the Web. On-line information streams encode information about both these kinds of data, and the division between the two is far from clear. While research paper archives and news streams clearly represent written content, the written material in weblogs is quite closely tied to the usage patterns of its authors—their recent history of reading and communication. The text that is posted on forums and discussion boards also reflects the dynamics of visitors to sites on a rapid time-scale; and search engine query logs also occupy a kind of middle ground, consisting of text encoding the behavior of searchers at the site.

To conclude our survey of different application areas, we consider an on-line information stream that encodes a basic form of usage data—the sequence of downloads performed by users at a high-traffic Web site. Many active Web sites are organized around a set of items that are available for purchase or download, using a fairly consistent high-level metaphor: at an e-commerce site like amazon.com, or an archive of research papers like arxiv.org, there is navigational structure at the front, followed by a large set of “description pages, one associated with each item that is available. Each description page provides the option to acquire the corresponding item.

In joint work with Jon Aizen, Dan Huttenlocher, and Tony Novak [3], the author studied the dynamics of download behavior at one such site, the Internet Archive, which maintains a publicly accessible media collection consisting of old films, live concerts, free on-line books, and other items available for download. The basic definition in [3] is the “batting average” (henceforth, the BA) of an on-line item: the number of acquisitions divided by the number of visits to the description page. Thus a high BA is a reflection of item quality, indicating that a large fraction of users who viewed the item’s description chose to download it. Note that this is different from a simple “hit counter” which measures the raw number of item visits or downloads; the BA addresses the more subtle notion of users’ reactions to the item description.

Suppose we want to measure fluctuations in the BA over time, looking for evidence of bursty behavior in which the BA of an item changes suddenly; in order to do this it is necessary to define a meaningful notion of the “instantaneous BA” of an item as a function of time. Such an instantaneous BA must be synthesized from the 0–1-valued sequence of user download decisions: each entry in the usage log simply reflects whether a particular chose to download the item or not. In [3], two different methods are considered for determining an instantaneous BA from this sequence. First, one can apply Gaussian smoothing, computing an average of each point in the download sequence with its neighbors, weighted by coefficients that decay according to a Gaussian function. Alternately, one can treat the download decisions as the 0–1-valued outputs of a Hidden Markov model (HMM), with hidden states representing underlying download probabilities; the maximum-likelihood state sequence can then be taken to represent the item’s BA at each point in time. A large number of states was used in the HMM computation, so as to densely sample the set of possible probabilities; in order to be able to handle long download sequences efficiently, this required the use of an improved maximum-likelihood algorithm due to Felzenszwalb et al. [14] that took advantage of the structure of the state transitions so as to run in time linear in the number of states, rather than quadratic. The HMM approach

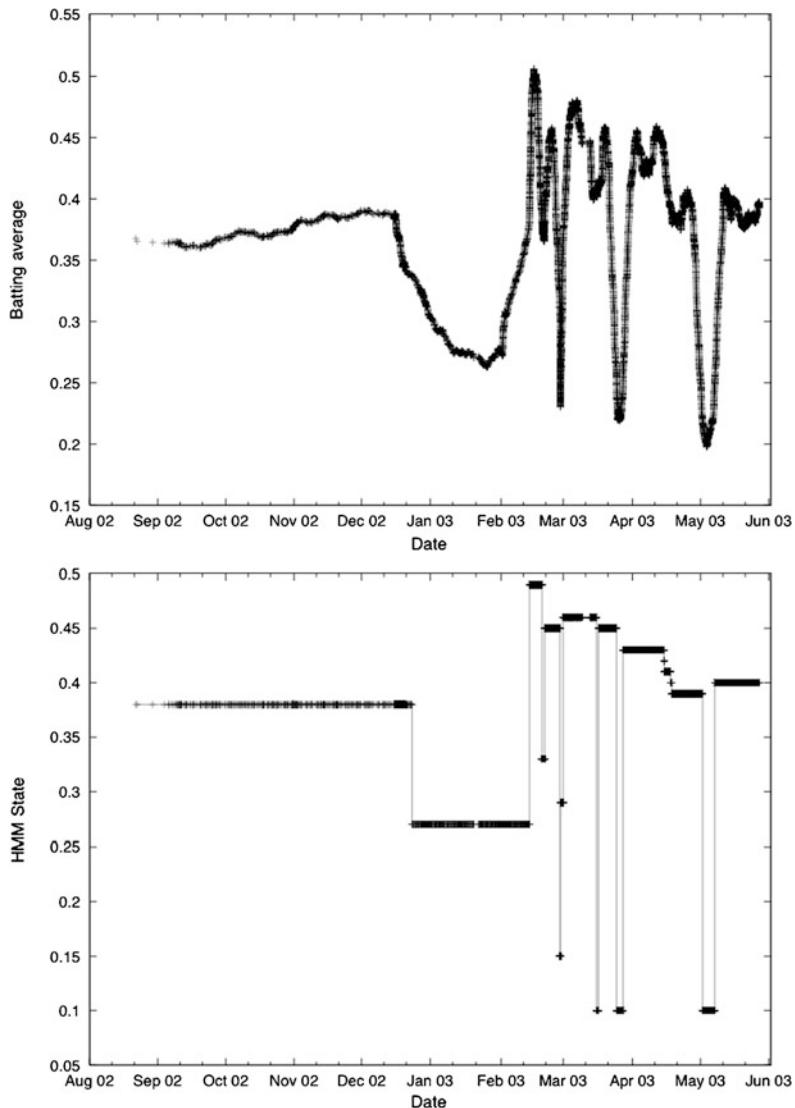


Fig. 1 Tracking the batting average of a downloadable movie at the Internet Archive as a function of time. The upper plot shows smoothing by averaging with a Gaussian sliding window; the lower plot shows the maximum-likelihood state sequence in a Hidden Markov model

produces sharp, discrete changes in the BA while the Gaussian smoothing approach yields more gradual changes; Fig. 1 illustrates this, showing the BA as a function of time for each of these approaches applied to the same downloadable movie at the Internet Archive. One can see how the sharp steps in the HMM plot approximately line up with the more gradual curves in the Gaussian-smoothed plot.

Are these sharp steps useful in the context of the underlying application? In [3], it is argued that the discrete breakpoints in the BA produced by the HMM in fact capture a crucial feature of popularity dynamics at sites like the Internet Archive. Specifically, the download properties of an item, as reflected in measures like the BA, often change abruptly rather than gradually, due to the appearance of an external link from some other high-traffic site—which suddenly drives in a new mix of users with new interests—or due to on-site highlighting—for example, featuring the item on a top-level page at the Internet Archive—which also raises the item’s visibility. The addition of links or promotional text happens at a discrete point in time, and experiments in [3] show that state transitions in the HMM align closely with the times when these events happen. To continue with the plot in Fig. 1, for example, the first two transitions in the HMM plot align with on-site highlighting performed by the Internet Archive, the next two sharp drops correspond to the appearance of links to the item’s description from high-traffic weblogs, and the final three drops corresponds to technical problems on the Internet Archive site. Examples like this, as well as the more systematic evaluation performed in [3], suggest how accurate tracking of changes to an item’s BA can help in understanding the set of events both on and off the site that affect the item’s popularity.

4 Conclusions

In studying the temporal dynamics of on-line information streams, an issue that arises in many contexts is the problem of *alignment*: we want to align the “virtual” events that we find in the stream with a parallel set of events taking place outside the stream, in the “real world.” One sees this, for example, in aligning bursts in a news stream or weblog index with current events; or in aligning interesting temporal dynamics in search engine query terms with seasonal patterns, upcoming holidays, names in the news, or any of a range of other factors that drive search behavior. It would be interesting to formalize this alignment problem more fully, so that it can be used both to refine the analysis of these streams and to better evaluate the analyses that are performed. We have discussed some concrete examples of steps in this direction above; these include the use of time-series matching techniques to compare temporal profiles of different search engine queries [36], and the alignment of item popularity with on-site highlighting and off-site referrers at the Internet Archive. Another interesting piece of work in this direction is that of Lavrenko et al. [25], which aligns financial news stories with the behavior of financial markets. They and others make the point that the “virtual events” in the stream may not only follow as consequences of real-world events, but may also influence them—as when an unexpected news story about a company leads to a measurable effect on the company’s stock price.

Relatively little work has been done on designing algorithms for the problems considered here in a “pure” streaming model of computation, where the data must be produced in one or a small number of passes with limited storage. For some of

the applications, it is not clear that the data requirements are substantial enough that the use of such a streaming model will be necessary, but for other applications, including the analysis of query logs and clickstreams, there is a clear opportunity for algorithms developed in this style. The work of Ben-David et al. [9] and Charikar et al. [10] take steps in this direction, considering different ways of measuring change for streaming data.

Another direction that would be interesting to consider further is the problem of predicting bursts and other temporal patterns, when the data must be processed in real-time rather than through retrospective analysis. How early into the emergence of a bursty news topic, for example, can one detect it and begin forming estimates of its basic shape? This is clearly related to the problem of general *time-series prediction*, though there is the opportunity here to use a significant amount of domain knowledge based on the content of the text information streams being studied. Gathering domain knowledge involves developing a more refined understanding of the types of temporal patterns that regularly occur in these kinds of information streams. We know, for example, that the temporal profiles of some search engine queries are periodic (corresponding to weekends or annual holidays, for example) while others look like isolated spikes; we know that some news stories build up to an expected event while others appear suddenly in response to an unexpected one; but we do not have a careful classification or taxonomy of the full range of such patterns. Clearly, this would be very useful in helping to recognize temporal patterns as they arise in on-line content.

Finally, it is worth noting that as the applications of these techniques focus not just on news stories and professionally published documents but on weblogs, e-mail, search engine queries, and browsing histories, they move into domains that are increasingly about individual behavior. Many of these large data streams are personal; their subject is *us*. And as we continue to develop applications that extract detailed information from these streams—media players that have a more accurate picture of your changing tastes in music than you do, or e-mail managers that encode detailed awareness of the set of people you've fallen out of touch with—we will ultimately have to deal not just with technical questions but with yet another dimension of the way in which information-aware devices interact with our daily lives.

References

1. E. Adar, L. Zhang, L.A. Adamic, R.M. Lukose, Implicit structure and the dynamics of blogspace. Workshop on the weblogging ecosystem, at the international WWW conference (2004)
2. R. Agrawal, R. Srikant, Mining sequential patterns, in *Proc. Intl. Conf. on Data Engineering* (1995)
3. J. Aizen, D. Huttenlocher, J. Kleinberg, A. Novak, Traffic-based feedback on the web. *Proc. Natl. Acad. Sci.* **101**(Suppl. 1), 5254–5260 (2004)
4. J. Allan (ed.), *Topic Detection and Tracking: Event Based Information Retrieval* (Kluwer Academic, Norwell, 2002)

5. J. Allan, J.G. Carbonell, G. Doddington, J. Yamron, Y. Yang, Topic detection and tracking pilot study: final report, in *Proc. DARPA Broadcast News Transcription and Understanding Workshop* (1998)
6. R. Allen, Timelines as information system interfaces, in *Proc. International Symposium on Digital Libraries* (1995)
7. D. Anick, D. Mitra, M. Sondhi, Stochastic theory of a data handling system with multiple sources. *Bell Syst. Tech. J.* **61** (1982)
8. J. Ask, Top searches at <http://static.wc.ask.com/docs/about/jeevesiq.html?o=0>
9. S. Ben-David, J. Gehrke, D. Kifer, Detecting change in data streams, in *Proc. 30th Intl. Conference on Very Large Databases (VLDB)* (2004)
10. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in *Proc. Intl. Colloq. on Automata Languages and Programming* (2002)
11. Daypop. <http://www.daypop.com>
12. F. Diaz, R. Jones, Using temporal profiles of queries for precision prediction, in *Proc. SIGIR Intl. Conf. on Information Retrieval* (2004)
13. A. Elwalid, D. Mitra, Effective bandwidth of general Markovian traffic sources and admission control of high speed networks. *IEEE Trans. Netw.* **1** (1993)
14. P. Felzenszwalb, D. Huttenlocher, J. Kleinberg, Fast algorithms for large-state-space HMMs with applications to web usage analysis, in *Advances in Neural Information Processing Systems (NIPS)*, vol. 16 (2003)
15. E. Gabrilovich, S. Dumais, E. Horvitz, NewsJunkie: providing personalized newsfeeds via analysis of information novelty, in *Proceedings of the Thirteenth International World Wide Web Conference* (2004)
16. Google. Zeitgeist at <http://www.google.com/press/zeitgeist.html>
17. D. Gruhl, R. Guha, D. Liben-Nowell, A. Tomkins, Information diffusion through blogspace, in *Proc. International WWW Conference* (2004)
18. D. Hand, H. Mannila, P. Smyth, *Principles of Data Mining* (MIT Press, Cambridge, 2001)
19. S. Havre, B. Hetzler, L. Nowell, ThemeRiver: visualizing theme changes over time, in *Proc. IEEE Symposium on Information Visualization* (2000)
20. D. Jensen, Personal communication (2002)
21. F.P. Kelly, Notes on effective bandwidths, in *Stochastic Networks: Theory and Applications*, ed. by F.P. Kelly, S. Zachary, I. Ziedins (Oxford University Press, London, 1996)
22. D. Kempe, J. Kleinberg, E. Tardos, Maximizing the spread of influence through a social network, in *Proc. 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (2003)
23. J. Kleinberg, Bursty and hierarchical structure in streams, in *Proc. 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (2002)
24. R. Kumar, J. Novak, P. Raghavan, A. Tomkins, On the bursty evolution of blogspace, in *Proc. International WWW Conference* (2003)
25. V. Lavrenko, M. Schmill, D. Lawrie, P. Ogilvie, D. Jensen, J. Allan, Mining of concurrent text and time series. KDD-2000 workshop on text mining (2000)
26. R. Liebscher, R. Belew, Lexical dynamics and conceptual change: analyses and implications for information retrieval. *Cogn. Sci. (Online)* **1** (2003)
27. K. Mane, K. Börner, Mapping topics and topic bursts in PNAS. *Proc. Natl. Acad. Sci.* **101**(Suppl. 1), 5287–5290 (2004)
28. H. Mannila, H. Toivonen, A.I. Verkamo, Discovering frequent episodes in sequences, in *Proc. Intl. Conf. on Knowledge Discovery and Data Mining* (1995)
29. N. Miller, P. Wong, M. Brewster, H. Foote, Topic islands: a wavelet-based text visualization system, in *Proc. IEEE Visualization* (1998)
30. R. Papka, On-line new event detection, clustering, and tracking. PhD thesis, Univ. Mass. Amherst (1999)
31. C. Plaisant, B. Milash, A. Rose, S. Widoff, B. Shneiderman, LifeLines: visualizing personal histories, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (1996)

32. L. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* **77** (1989)
33. R. Swan, J. Allan, Extracting significant time-varying features from text, in *Proc. 8th Int'l. Conf. on Information Knowledge Management* (1999)
34. R. Swan, J. Allan, Automatic generation of overview timelines, in *Proc. SIGIR Int'l. Conf. on Information Retrieval* (2000)
35. R. Swan, D. Jensen, TimeMines: constructing timelines with statistical models of word usage. *KDD-2000 Workshop on Text Mining* (2000)
36. M. Vlachos, C. Meek, Z. Vagena, D. Gunopulos, Identifying similarities, periodicities and bursts for online search queries, in *Proc. ACM SIGMOD International Conference on Management of Data* (2004)
37. P. Wong, W. Cowley, H. Foote, E. Jurrus, J. Thomas, Visualizing sequential patterns for text mining, in *Proc. IEEE Information Visualization* (2000)
38. Y. Zhu, D. Shasha, Efficient elastic burst detection in data streams, in *Proc. ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (2003)

Part III

Advanced Topics

Sketch-Based Multi-Query Processing over Data Streams

Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi

We consider the problem of approximately answering multiple general *aggregate* SQL queries over continuous data streams with limited memory. Our method extends the randomizing techniques of Alon et al. [2] that compute small “sketch” summaries of the streams that can then be used to provide approximate answers to aggregate queries with provable guarantees on the approximation error. By intelligently sharing the “sketches” among multiple queries, the memory required can be reduced. We provide necessary and sufficient conditions for the sketch sharing to result in correct estimation and address optimization problems that arise. We also demonstrate how existing statistical information on the base data (e.g., histograms) can be used in the proposed framework to improve the quality of the approximation provided by our algorithms. The key idea is to intelligently partition the domain of the underlying attribute(s) and, thus, decompose the sketching problem in a way that provably tightens our guarantees.

A. Dobra (✉)

Department of Computer Science, University of Florida, Gainesville FL, USA
e-mail: adobra@cise.ufl.edu

M. Garofalakis

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

J. Gehrke

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA
e-mail: johannes@microsoft.com

R. Rastogi

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

1 Introduction

As apparent from the developments in the previous chapters, the strong incentive behind data-stream computation has given rise to several (theoretical and practical) studies of on-line or one-pass algorithms with limited memory requirements for different problems; examples include quantile and order-statistics computation [15, 20], estimating frequency moments and join sizes [2, 3], data clustering and decision-tree construction [10, 17], estimating correlated aggregates [12], and computing one-dimensional (i.e., single-attribute) histograms and Haar wavelet decompositions [14, 16]. Other related studies have proposed techniques for incrementally maintaining equi-depth histograms [13] and Haar wavelets [21], maintaining samples and simple statistics over sliding windows [7], as well as general, high-level architectures for stream database systems [4].

None of the earlier research efforts has addressed the general problem of processing general, possibly multi-join, aggregate queries over continuous data streams. On the other hand, efficient approximate multi-join processing has received considerable attention in the context of *approximate query answering*, a very active area of database research in recent years [1, 6, 11, 18, 19, 24]. The vast majority of existing proposals, however, rely on the assumption of a static data set which enables either several passes over the data to construct effective, *multi-dimensional* data synopses (e.g., histograms [19] and Haar wavelets [6, 24]) or intelligent strategies for randomizing the access pattern of the relevant data items [18]. When dealing with continuous data streams, it is crucial that the synopsis structure(s) are constructed directly on the stream, that is, in one pass over the data in the fixed order of arrival; this requirement renders conventional approximate query processing tools inapplicable in a data-stream setting. (Note that, even though random-sample data summaries can be easily constructed in a single pass [23], it is well known that such summaries typically give very poor result estimates for queries involving one or more joins [1, 2, 6]¹).

In this chapter, we tackle the hard technical problems involved in the approximate processing of complex (possibly multi-join) aggregate decision-support queries over continuous data streams with limited memory. Our approach is based on randomizing techniques that compute small, pseudo-random *sketch* summaries of the data as it is streaming by. The basic sketching technique was originally introduced for on-line self-join size estimation by Alon, Matias, and Szegedy in their seminal paper [3] and, as we demonstrate in our work, can be generalized to provide approximate answers to complex, multi-join, aggregate SQL queries over streams with explicit and tunable guarantees on the approximation error. We should point out that the error-bound derivation for multi-join queries is non-trivial and requires that certain acyclicity restrictions be imposed on the query's join graph.

Usually, in a realistic application, multiple queries need to be processed simultaneously over a collection of data-streams. In this chapter, we also address the problem of efficiently sharing and allocating memory for the concurrent queries—we

¹The sampling-based *join synopses* of [1] provide a solution to this problem but only for the special case of *static, foreign-key joins*.

call our technique *sketch sharing*. We provide necessary and sufficient conditions for multi-query sketch sharing that guarantee correctness of the resulting sketch-based estimators. Since the problem of finding the optimal sketch sharing and memory allocation turns out to be \mathcal{NP} -hard, we provide a greedy algorithm to find reasonably good solutions.

Another important practical concern that arises in the multi-join context is that the quality of the approximation may degrade as the variance of our randomized sketch synopses increases in an explosive manner with the number of joins involved in the query. To this end, we propose novel *sketch-partitioning* techniques that take advantage of existing approximate statistical information on the stream (e.g., histograms built on archived data) to decompose the sketching problem in a way that provably tightens our estimation guarantees.

2 Answering a Single Multi-Join Query

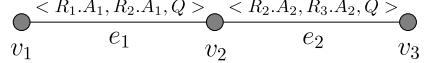
We begin by describing our sketch-based techniques to approximate the result of a single *multi-join* aggregate SQL query over a collection of streaming relations R_1, \dots, R_r . More specifically, we focus on approximating a multi-join stream query Q of the form: “SELECT COUNT FROM R_1, R_2, \dots, R_r WHERE \mathcal{E} ”, where \mathcal{E} represents the conjunction of n equi-join constraints of the form $R_i.A_j = R_k.A_l$ ($R_i.A_j$ denotes the j th attribute of relation R_i , and we use $\text{dom}(R_i.A_j)$ to denote its domain²). The extension to other aggregate functions, e.g., SUM, is fairly straightforward [9]; furthermore, note that dealing with single-relation selections is similarly straightforward (simply filter out the tuples that fail the selection predicate from the relational stream).

Our development also assumes that each attribute $R_i.A_j$ appears in \mathcal{E} at most once; this requirement can be easily achieved by simply renaming repeating attributes in the query. In what follows, we describe the key ideas and results based on the join-graph model of the input query Q , since this will allow for a smoother transition to the multi-query case (Sect. 3).

Given stream query Q , we define the *join graph* of Q (denoted by $\mathcal{J}(Q)$), as follows. There is a distinct vertex v in $\mathcal{J}(Q)$ for each stream R_i referenced in Q (we use $R(v)$ to denote the relation associated with vertex v). For each equality constraint $R_i.A_j = R_k.A_l$ in \mathcal{E} , we add a distinct undirected edge $e = \langle v, w \rangle$ to $\mathcal{J}(Q)$, where $R(v) = R_i$ and $R(w) = R_k$; we also label this edge with the triple $\langle R_i.A_j, R_k.A_l, Q \rangle$ that specifies the attributes in the corresponding equality constraint and the enclosing query Q (the query label is used in the multi-query setting). Given an edge $e = \langle v, w \rangle$ with label $\langle R_i.A_j, R_k.A_l, Q \rangle$, the three components of e 's label triple can be obtained as $A_v(e)$, $A_w(e)$ and $Q(e)$. (Clearly, by the definition of equi-joins, $\text{dom}(A_v(e)) = \text{dom}(A_w(e))$.) Note that there may be multiple edges

²Without loss of generality, we assume that each attribute domain $\text{dom}(A)$ is indexed by the set of integers $\{1, \dots, |\text{dom}(A)|\}$, where $|\text{dom}(A)|$ denotes the size of the domain.

Fig. 1 Example query join graph



between a pair of vertices in the join graph, but each edge has its own distinct label triple. Finally, for a vertex v in $\mathcal{J}(Q)$, we denote the attributes of $R(v)$ that appear in the input query (or, queries) as $\mathcal{A}(v)$; thus, $\mathcal{A}(v) = \{A_v(e) : \text{edge } e \text{ is incident on } v\}$.

The result of Q is the number of tuples in the cross-product of R_1, \dots, R_r that satisfy the equality constraints in \mathcal{E} over the join attributes. Similar to the basic sketching method [2, 3], our algorithm constructs an unbiased, bounded-variance probabilistic estimate X_Q for Q using atomic sketches built on the vertices of the join graph $\mathcal{J}(Q)$. More specifically, for each edge $e = \langle v, w \rangle$ in $\mathcal{J}(Q)$, our algorithm defines a family of four-wise independent random variables $\xi^e = \{\xi_i^e : i = 1, \dots, |\text{dom}(A_v(e))|\}$, where each $\xi_i^e \in \{-1, +1\}$. The key here is that the equi-join attribute pair $A_v(e), A_w(e)$ associated with edge e shares the same ξ family; on the other hand, distinct edges of $\mathcal{J}(Q)$ use *independently-generated* ξ families (using mutually independent random seeds). The atomic sketch X_v for each vertex v in $\mathcal{J}(Q)$ is built as follows. Let e_1, \dots, e_k be the edges incident on v and, for $i_1 \in \text{dom}(A_v(e_1)), \dots, i_k \in \text{dom}(A_v(e_k))$, let $f_v(i_1, \dots, i_k)$ denote the number of tuples in $R(v)$ that match values i_1, \dots, i_k in their join attributes. More formally, $f_v(i_1, \dots, i_k)$ is the number of tuples $t \in R(v)$ such that $t[A_v(e_j)] = i_j$, for $1 \leq j \leq k$ ($t[A]$ denotes the value of attribute A in tuple t). Then, the atomic sketch at v is $X_v = \sum_{i_1 \in \text{dom}(A_v(e_1))} \dots \sum_{i_k \in \text{dom}(A_v(e_k))} f_v(i_1, \dots, i_k) \prod_{j=1}^k \xi_{i_j}^{e_j}$. Finally, the estimate for Q is defined as $X_Q = \prod_v X_v$ (that is, the product of the atomic sketches for all vertices in $\mathcal{J}(Q)$). Note that each atomic sketch X_v is again a randomized linear projection that can be efficiently computed as tuples of $R(v)$ are streaming in; more specifically, X_v is initialized to 0 and, for each tuple t in the $R(v)$ stream, the quantity $\prod_{j=1}^k \xi_{t[A_v(e_j)]}^{e_j} \in \{-1, +1\}$ is added to X_v .

Example 1 Consider query $Q = \text{SELECT COUNT FROM } R_1, R_2, R_3 \text{ WHERE } R_1.A_1 = R_2.A_1 \text{ AND } R_2.A_2 = R_3.A_2$. The join graph $\mathcal{J}(Q)$ is depicted in Fig. 1, with vertices v_1 , v_2 , and v_3 corresponding to streams R_1 , R_2 , and R_3 , respectively. Similarly, edges e_1 and e_2 correspond to the equi-join constraints $R_1.A_1 = R_2.A_1$ and $R_2.A_2 = R_3.A_2$, respectively. (Just to illustrate our notation, $R(v_1) = R_1$, $A_{v_2}(e_1) = R_2.A_1$ and $\mathcal{A}(v_2) = \{R_2.A_1, R_2.A_2\}$.) The sketch construction defines two families of four-wise independent random families (one for each edge): $\{\xi_i^{e_1}\}$ and $\{\xi_j^{e_2}\}$. The three atomic sketches X_{v_1} , X_{v_2} , and X_{v_3} (one for each vertex) are defined as: $X_{v_1} = \sum_{i \in \text{dom}(R_1.A_1)} f_{v_1}(i) \xi_i^{e_1}$, $X_{v_2} = \sum_{i \in \text{dom}(R_2.A_1)} \sum_{j \in \text{dom}(R_2.A_2)} f_{v_2}(i, j) \xi_i^{e_1} \xi_j^{e_2}$, and $X_{v_3} = \sum_{j \in \text{dom}(R_3.A_2)} f_{v_3}(j) \xi_j^{e_2}$. The value of random variable $X_Q = X_{v_1} X_{v_2} X_{v_3}$ gives the sketching estimate for the result of Q .

Our analysis in [9] showed that the random variable X_Q constructed above is an unbiased estimator for Q , and demonstrates the following theorem which generalizes the earlier result of Alon et al. [2] to multi-join queries. ($SJ_v = \sum_{i_1 \in \text{dom}(A_v(e_1))} \dots \sum_{i_k \in \text{dom}(A_v(e_k))} f_v(i_1, \dots, i_k)^2$ is the self-join size of $R(v)$.)

Theorem 1 ([9]) Let Q be a COUNT query with n equi-join predicates such that $\mathcal{J}(Q)$ contains no cycles of length > 2 . Then, $E[X_Q] = Q$ and using sketching space of $O(\frac{\text{Var}[X_Q] \cdot \log(1/\delta)}{Q^2 \cdot \epsilon^2})$, it is possible to approximate Q to within a relative error of ϵ with probability at least $1 - \delta$, where $\text{Var}[X_Q] \leq 2^{2n} \prod_v \text{SJ}_v$.

3 Answering Multiple Join Queries: Sketch Sharing

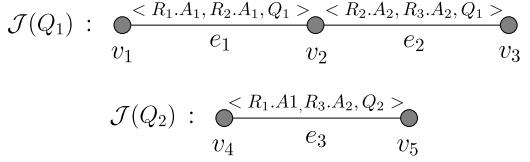
We next turn our attention to sketch-based processing of *multiple* aggregate SQL queries over streams. We introduce the basic idea of sketch sharing and demonstrate how it can improve the effectiveness of the available sketching space and the quality of the resulting approximate answers. We also characterize the class of correct sketch-sharing configurations and formulate the optimization problem of identifying an effective sketch-sharing plan for a given query workload. We finally propose a greedy heuristic for computing good sketch-sharing configurations with minimal space overhead.

3.1 Sketch Sharing: Basic Concept

Consider the problem of using sketch synopses for the effective processing of a query workload $Q = \{Q_1, \dots, Q_q\}$ comprising multiple (multi-join) COUNT aggregate queries. As in the previous section, we focus on COUNT since the extension to other aggregate functions is relatively straightforward; we also assume an attribute-renaming step that ensures that each stream attribute is referenced only once in each of the Q_i 's (of course, the same attribute can be used multiple times across the queries in Q). Finally, as before, we do not consider single-relation selections, since they can be trivially incorporated in the model by using the selection predicates to define filters for each stream. The sketching of each relation is performed using only the tuples that pass the filter; this is equivalent to introducing virtual relations/streams that are the result of the filtering process and formulating the queries with respect to these relations. This could potentially increase the number of relations and reduce the number of opportunities to share sketches (as described in this section), but would also create opportunities similar to the ones investigated by traditional MQO (e.g., constructing sketches for common filter sub-expressions). Here, we focus on the novel problem of sharing sketches and we do not investigate further how such techniques can be used for the case where selection predicates are allowed. As will become apparent in this section, sketch sharing is very different from common sub-expression sharing; traditional MQO techniques do not apply for this problem.

An obvious solution to our multi-query processing problem is to build disjoint join graphs $\mathcal{J}(Q_i)$ for each query $Q_i \in Q$, and construct independent atomic sketches for the vertices of each $\mathcal{J}(Q_i)$. The atomic sketches for each vertex of

Fig. 2 Example workload with sketch-sharing potential



$\mathcal{J}(Q_i)$ can then be combined to compute an approximate answer for Q_i as described in Sect. 2. A key drawback of such a naive solution is that it ignores the fact that a relation R_i may appear in multiple queries in \mathcal{Q} . Thus, it should be possible to reduce the overall space requirements by *sharing* atomic-sketch computations among the vertices for stream R_i in the join graphs for the queries in our workload. We illustrate this in the following example.

Example 2 Consider queries $Q_1 = \text{SELECT COUNT FROM } R_1, R_2, R_3 \text{ WHERE } R_1.A_1 = R_2.A_1 \text{ AND } R_2.A_2 = R_3.A_2$ and $Q_2 = \text{SELECT COUNT FROM } R_1, R_3 \text{ WHERE } R_1.A_1 = R_3.A_2$. The naive processing algorithm described above would maintain two disjoint join graphs (Fig. 2) and, to compute a single pair (X_{Q_1}, X_{Q_2}) of sketch-based estimates, it would use three families of random variables $(\xi^{e_1}, \xi^{e_2}, \text{ and } \xi^{e_3})$, and a total of five atomic sketches $(X_{v_k}, k = 1, \dots, 5)$.

Instead, suppose that we decide to re-use the atomic sketch X_{v_1} for v_1 also for v_4 , both of which essentially correspond to the same attribute of the same stream ($R_1.A_1$). Since for each $i \in \text{dom}(R_1.A_1)$, $f_{v_4}(i) = f_{v_1}(i)$, we get $X_{v_4} = X_{v_1} = \sum_{i \in \text{dom}(R_1.A_1)} f_{v_4}(i) \xi_i^{e_1}$. Of course, in order to correctly compute a probabilistic estimate of Q_2 , we also need to use the same family ξ^{e_1} in the computation of X_{v_5} ; that is, $X_{v_5} = \sum_{i \in \text{dom}(R_1.A_1)} f_{v_5}(i) \xi_i^{e_1}$. It is easy to see that both final estimates $X_{Q_1} = X_{v_1} X_{v_2} X_{v_3}$ and $X_{Q_1} = X_{v_1} X_{v_5}$ satisfy all the premises of the sketch-based estimation results in [9]. Thus, by simply sharing the atomic sketches for v_1 and v_4 , we have reduced the total number of random families used in our multi-query processing algorithm to two (ξ^{e_1} and ξ^{e_2}) and the total number of atomic sketches maintained to four.

Let $\mathcal{J}(\mathcal{Q})$ denote the collection of all join graphs in workload \mathcal{Q} , i.e., all $\mathcal{J}(Q_i)$ for $Q_i \in \mathcal{Q}$. Sharing sketches between the vertices of $\mathcal{J}(\mathcal{Q})$ can be seen as a transformation of $\mathcal{J}(\mathcal{Q})$ that essentially *coalesces* vertices belonging to different join graphs in $\mathcal{J}(\mathcal{Q})$. (We also use $\mathcal{J}(\mathcal{Q})$ to denote the transformed multi-query join graph.) Of course, as shown in Example 2, vertices $v \in \mathcal{J}(Q_i)$ and $w \in \mathcal{J}(Q_j)$ can be coalesced in this manner *only if* $R(v) = R(w)$ (i.e., they correspond to the same data stream) and $\mathcal{A}(v) = \mathcal{A}(w)$ (i.e., both Q_i and Q_j use exactly the same attributes of that stream). Such vertex coalescing implies that a vertex v in $\mathcal{J}(\mathcal{Q})$ can have edges from multiple different queries incident on it; we denote the set of all these queries as $Q(v)$, i.e., $Q(v) = \{Q(e) : \text{edge } e \text{ is incident on } v\}$. Figure 3(a) pictorially depicts the coalescing of vertices v_1 and v_4 as discussed in Example 2. Note that, by our coalescing rule, for each vertex v , all queries in $Q(v)$ are guaranteed to use exactly the same set of attributes of $R(v)$, namely $\mathcal{A}(v)$; furthermore, by our attribute-renaming step, each query in $Q(v)$ uses each attribute in $\mathcal{A}(v)$ exactly

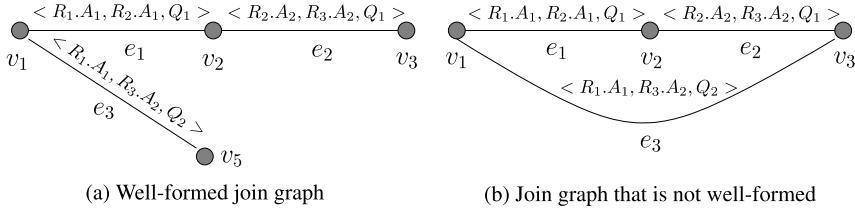


Fig. 3 Multi-query join graphs $\mathcal{J}(\mathcal{Q})$ for Example 2

once. This makes it possible to share an atomic sketch built for the coalesced vertices v across all queries in $Q(v)$ but, as we will see shortly, cannot guarantee the correctness of the resulting sketch-based estimates.

Estimation with Sketch Sharing

Consider a multi-query join graph $\mathcal{J}(\mathcal{Q})$, possibly containing coalesced vertices (as described above). Our goal here is to build *atomic sketches corresponding to individual vertices* of $\mathcal{J}(\mathcal{Q})$ that can then be used for obtaining sketch-based estimates for *all* the queries in our workload \mathcal{Q} . Specifically, consider a query $Q \in \mathcal{Q}$, and let $V(Q)$ denote the (sub)set of vertices in $\mathcal{J}(\mathcal{Q})$ attached to a join-predicate edge corresponding to Q ; that is, $V(Q) = \{v : \text{edge } e \text{ is incident on } v \text{ and } Q(e) = Q\}$. Our goal is to construct an unbiased probabilistic estimate X_Q for Q using the atomic sketches built for vertices in $V(Q)$.

The atomic sketch for a vertex v of $\mathcal{J}(\mathcal{Q})$ is constructed as follows. As before, each edge $e \in \mathcal{J}(\mathcal{Q})$ is associated with a family ξ^e of four-wise independent $\{-1, +1\}$ random variables. The difference here, however, is that edges attached to node v for the *same attribute* of $R(v)$ share the *same* ξ family since the *same* sketch of $R(v)$ corresponding to vertex v is used to estimate *all* queries in $Q(v)$; this, of course, implies that the number of *distinct* ξ families for all edges incident on v is exactly $|\mathcal{A}(v)|$ (each family corresponding to a distinct attribute of $R(v)$). Furthermore, all distinct ξ families in $\mathcal{J}(\mathcal{Q})$ are generated independently (using mutually independent seeds). For example, in Fig. 3(a), since $A_{v_1}(e_1) = A_{v_1}(e_3) = R_1.A_1$, edges e_1 and e_3 share the same ξ family (i.e., $\xi^{e_3} = \xi^{e_1}$); on the other hand, ξ^{e_1} and ξ^{e_2} are distinct and independent. Assuming $\mathcal{A}(v) = \{A_1, \dots, A_k\}$ and letting ξ^1, \dots, ξ^k denote the k corresponding distinct ξ families attached to v , the atomic sketch X_v for node v is simply defined as $X_v = \sum_{(i_1, \dots, i_k) \in A_1 \times \dots \times A_k} f_v(i_1, \dots, i_k) \prod_{j=1}^k \xi_{i_j}^j$ (again, a randomized linear projection). The final sketch-based estimate for query Q is the product of the atomic sketches over all vertices in $V(Q)$, i.e., $X_Q = \prod_{v \in V(Q)} X_v$.

Correctness of Sketch-Sharing Configurations

The X_Q estimate construction described above can be viewed as simply “extracting” the join (sub)graph $\mathcal{J}(Q)$ for query Q from the multi-query graph $\mathcal{J}(\mathcal{Q})$, and constructing a sketch-based estimate for Q as described in Sect. 2. This is because, if we were to only retain in $\mathcal{J}(\mathcal{Q})$ vertices and edges associated with Q , then the resulting subgraph is identical to $\mathcal{J}(Q)$. Furthermore, our vertex coalescing (which completely determines the sketches to be shared) guarantees that Q references exactly the attributes $\mathcal{A}(v)$ of $R(v)$ for each $v \in V(Q)$, so the atomic sketch X_v can be utilized.

There is, however, an important complication that our vertex-coalescing rule still needs to address, to ensure that the atomic sketches for vertices of $\mathcal{J}(\mathcal{Q})$ provide unbiased query estimates with variance bounded as described in Theorem 1. Given an estimate X_Q for query Q (constructed as above), unbiasedness and the bounds on $\text{Var}[X_Q]$ given in Theorem 1 depend crucially on the assumption that the ξ families used for the edges in $\mathcal{J}(Q)$ are distinct and independent. This means that simply coalescing vertices in $\mathcal{J}(\mathcal{Q})$ that use the same set of stream attributes is insufficient. The problem here is that the constraint that all edges for the same attribute incident on a vertex v share the same ξ family may (by transitivity) force edges for the same query Q to share identical ξ families. The following example illustrates this situation.

Example 3 Consider the multi-query join graph $\mathcal{J}(\mathcal{Q})$ in Fig. 3(b) for queries Q_1 and Q_2 in Example 3. ($\mathcal{J}(\mathcal{Q})$ is obtained as a result of coalescing vertex pairs v_1, v_4 and v_3, v_5 in Fig. 2.) Since $A_{v_1}(e_1) = A_{v_1}(e_3) = R_1.A_1$ and $A_{v_3}(e_2) = A_{v_3}(e_3) = R_3.A_2$, we get the constraints $\xi^{e_3} = \xi^{e_1}$ and $\xi^{e_3} = \xi^{e_2}$. By transitivity, we have $\xi^{e_1} = \xi^{e_2} = \xi^{e_3}$, i.e., all three edges of the multi-query graph share the same ξ family. This, in turn, implies that the same ξ family is used on both edges of query Q_1 ; that is, instead of being independent, the pseudo-random families used on the two edges of Q_1 are perfectly correlated! It is not hard to see that, in this situation, the expectation and variance derivations for X_{Q_1} will fail to produce the results of Theorem 1, since many of the zero cross-product terms in the analysis of [2, 9] will fail to vanish.

As is clear from the above example, the key problem is that constraints requiring ξ families for certain edges incident on each vertex of $\mathcal{J}(\mathcal{Q})$ to be identical, can transitively ripple through the graph, forcing much larger sets of edges to share the same ξ family. We formalize this fact using the following notion of (transitive) ξ -equivalence among edges of a multi-query graph $\mathcal{J}(\mathcal{Q})$.

Definition 1 Two edges e_1 and e_2 in $\mathcal{J}(\mathcal{Q})$ are said to be ξ -equivalent if either (i) e_1 and e_2 are incident on a common vertex v , and $A_v(e_1) = A_v(e_2)$; or (ii) there exists an edge e_3 such that e_1 and e_3 are ξ -equivalent, and e_2 and e_3 are ξ -equivalent.

Intuitively, the classes of the ξ -equivalence relation represent exactly the sets of edges in the multi-query join graph $\mathcal{J}(\mathcal{Q})$ that need to share the same ξ family; that

is, for any pair of ξ -equivalent edges e_1 and e_2 , it is the case that $\xi^{e_1} = \xi^{e_2}$. Since, for estimate correctness, we require that all the edges associated with a query have distinct and independent ξ families, our sketch-sharing algorithms only consider multi-query join graphs that are *well-formed*, as defined below.

Definition 2 A multi-query join graph $\mathcal{J}(Q)$ is *well-formed* iff, for every pair of ξ -equivalent edges e_1 and e_2 in $\mathcal{J}(Q)$, the queries containing e_1 and e_2 are distinct, i.e., $Q(e_1) \neq Q(e_2)$.

It is not hard to prove that the well-formedness condition described above is actually necessary and sufficient for individual sketch-based query estimates that are unbiased and obey the variance bounds of Theorem 1. Thus, our shared-sketch estimation process over well-formed multi-query graphs can readily apply the single-query results of [2, 9] for each individual query in our workload.

3.2 Sketch Sharing Problem Formulation

Given a large workload Q of complex queries, there can obviously be a large number of well-formed join graphs for Q , and all of them can potentially be used to provide approximate sketch-based answers to queries in Q . At the same time, since the key resource constraint in a data-streaming environment is imposed by the amount of memory available to the query processor, our objective is to compute approximate answers to queries in Q that are as accurate as possible given a fixed amount of memory M for the sketch synopses. Thus, in the remainder of this chapter, we focus on the problem of computing (i) a well-formed join graph $\mathcal{J}(Q)$ for Q , and (ii) an allotment of the M units of space to the vertices of $\mathcal{J}(Q)$ (for maintaining iid copies of atomic sketches), such that an appropriate aggregate error metric (e.g., average or maximum error) for all queries in Q is minimized.

More formally, let m_v denote the sketching space allocated to vertex v (i.e., number of iid copies of X_v). Also, let M_Q denote the number of iid copies built for the query estimate X_Q . Since $X_Q = \prod_{v \in V(Q)} X_v$, it is easy to see that M_Q is actually constrained by the *minimum* number of iid atomic sketches constructed for each of the nodes in $V(Q)$; that is, $M_Q = \min_{v \in V(Q)} \{m_v\}$. By Theorem 1, this implies that the (square) error for query Q is equal to W_Q/M_Q , where $W_Q = \frac{8\text{Var}[X_Q]}{E[X_Q]^2}$ is a constant for each query Q (assuming a fixed confidence parameter δ). Our sketch-sharing optimization problem can then be formally stated as follows.

Problem Statement

Given a query workload $Q = \{Q_1, \dots, Q_q\}$ and an amount of sketching memory M , compute a multi-query graph $\mathcal{J}(Q)$ and a space allotment $\{m_v : v \in V(\mathcal{J}(Q))\}$ such that one of the following two error metrics is minimized:

- Average query error in $\mathcal{Q} = \sum_{Q \in \mathcal{Q}} \frac{W_Q}{M_Q}$,
- Maximum query error in $\mathcal{Q} = \max_{Q \in \mathcal{Q}} \{\frac{W_Q}{M_Q}\}$,

subject to the constraints: (i) $\mathcal{J}(\mathcal{Q})$ is well-formed; (ii) $\sum_v m_v \leq M$ (i.e., the space constraint is satisfied); and (iii) For all vertices v in $\mathcal{J}(\mathcal{Q})$, for all queries $Q \in Q(v)$, $M_Q \leq m_v$.

The above problem statement assumes that the “weight” W_Q for each query $Q \in \mathcal{Q}$ is known. Clearly, if coarse statistics in the form of histograms for the stream relations are available (e.g., based on historical information or coarse a-priori knowledge of data distributions), then estimates for $E[X_Q]$ and $\text{Var}[X_Q]$ (and, consequently, W_Q) can be obtained by estimating join and self-join sizes using these histograms [9]. In the event that no prior information is available, we can simply set each $W_Q = 1$; unfortunately, even for this simple case, our optimization problem is intractable (see Sect. 3.3).

In the following subsection, we first consider the sub-problem of optimally allocating sketching space (such that query errors are minimized) to the vertices of a given, well-formed join graph $\mathcal{J}(\mathcal{Q})$. Subsequently, in Sect. 3.4, we consider the general optimization problem where we also seek to determine the best well-formed multi-query graph for the given workload \mathcal{Q} . Since most of these questions turn out to be \mathcal{NP} -hard, we propose novel heuristic algorithms for determining good solutions in practice. Our algorithm for the overall problem (Sect. 3.4) is actually an iterative procedure that uses the space-allocation algorithms of Sect. 3.3 as subroutines in the search for a good sketch-sharing plan.

3.3 Space Allocation Problem

In this subsection, we consider the problem of allocating space optimally given a well-formed join graph $J = \mathcal{J}(\mathcal{Q})$ such that the average or maximum error is minimized.

Minimizing the Average Error

The problem of allocating space to sketches in a way that minimizes the average error turns out to be \mathcal{NP} -hard even when $W_Q = 1$. Given the intractability of the problem, we look for an approximation based on its continuous relaxation, i.e., we allow the M_Q ’s and m_v ’s to be continuous. The continuous version of the problem is a convex optimization problem, which can be solved exactly in polynomial time using, for example, interior point methods [22]. We can then show that a near-optimal integer solution is obtained by rounding down (to integers) the optimal continuous values of the M_Q ’s and m_v ’s.

Since standard methods for solving convex optimization problems tend to be slow in practice, we developed a novel specialized solution for the problem at hand.

Our solution, which we believe has applications to a much wider class of problems than the optimal space allocation problem outlined here, is based on a novel usage of the Kuhn–Tucker optimality conditions (KT-conditions). We rewrite the problem using the KT conditions, and then we solve the problem through repeated application of a specific Max-Flow formulation of the constraints. Due to space limitations, we omit a detailed description of the algorithm and the analysis of its correctness (see [8] for details). Our results are summarized in the following theorem:

Theorem 2 *There is an algorithm that computes the optimal solution to the average-error continuous convex optimization problem in at most $O(\min\{|\mathcal{Q}|, |J|\} \cdot (|\mathcal{Q}| + |J|)^3)$ steps. Furthermore, rounding this optimal continuous solution results in an integer solution that is guaranteed to be within a factor of $(1 + \frac{2|J|}{M})$ of the optimal integer solution.*

Minimizing the Maximum Error

It can be easily shown (see [8] for details) that the problem of minimizing the maximum error can be solved in time $O(|J| \log |J|)$ by the following greedy algorithm: (i) take each m_v proportional to $\max_{Q \in \mathcal{Q}(v)} \{W_Q\}$, (ii) round down each m_v component to the nearest integer, and (iii) take the remaining space $s \leq |J|$ and allocate one extra unit of space to each of the nodes with the s smallest values for quantity $m_v / \max_{Q \in \mathcal{Q}(v)} \{W_Q\}$.

3.4 Computing a Well-Formed Join Graph

The optimization problem we are trying to solve is finding a well-formed graph $\mathcal{J}(\mathcal{Q})$ and the optimal space allocation to the vertices of $\mathcal{J}(\mathcal{Q})$ such that the average or maximum error is minimized. If we take $W_Q = 1$ for all queries and minimize the maximum error, this optimization problem reduces to the problem of finding a well-formed join graph $\mathcal{J}(\mathcal{Q})$ with the minimum number of vertices; this problem is \mathcal{NP} -hard (see [8] for the proof) which makes the initial optimization problem \mathcal{NP} -hard as well.

In order to provide an acceptable solution in practice we designed a greedy heuristic, that we call `CoalesceJoinGraphs`, for computing a well-formed join graph with small error. The Algorithm `CoalesceJoinGraphs` iteratively merges pair of vertices in J that causes the largest decrease in error, until the error cannot be reduced any further by coalescing vertices. It uses the algorithm to compute the average or maximum error (Sect. 3.3) for a join graph as a subroutine, which we denote by `ComputeSpace`, at every step. Also, in order to ensure that graph J always stays well-formed, J is initially set to be equal to the set of all the individual join graphs for queries in \mathcal{Q} . In each subsequent iteration, only vertices for identical relations that have the same attribute sets and preserve the well-formedness of J are

coalesced. Well-formedness testing essentially involves partitioning the edges of J into equivalence classes, each class consisting of ξ -equivalent edges, and then verifying that no equivalence class contains multiple edges from the same join query; it can be carried out very efficiently, in time proportional to the number of edges in J . The Algorithm `CoalesceJoinGraphs` needs to make at most $O(N^3)$ calls to `ComputeSpace`, where N is the total number of vertices in all the join graphs $\mathcal{J}(Q)$ for the queries, and this determines its time complexity.

4 Improving Answer Quality: Sketch Partitioning

In the proof of Theorem 1, to ensure (with high probability) an upper bound of ϵ on the relative error of our estimate for a COUNT query Q with n equi-join predicates, we require sketching space proportional to $\text{Var}[X_Q]$, where $\text{Var}[X_Q] \leq 2^{2n} \prod_v \text{SJ}_v$. (Recall from Sect. 2 that SJ_v is the self-join size of the relation $R(v)$ corresponding to vertex v in $\mathcal{J}(Q)$.) Thus, an important practical concern for multi-join queries is that our upper bound on the $\text{Var}[X_Q]$ and, therefore, the storage space required to guarantee a given level of accuracy increases explosively with the number of joins n in the query.

To deal with this problem, in this section, we propose novel *sketch-partitioning* techniques that exploit approximate statistics on the streams to decompose the sketching problem in a way that provably tightens our estimation guarantees. The basic idea is that, by intelligently partitioning the domain of join attributes in the query and estimating portions of Q individually on each partition, we can significantly reduce the storage required to compute each query estimate X_Q within a given level of accuracy. (Of course, our sketch-partitioning results are equally applicable to the dual optimization problem; that is, maximizing the estimation accuracy for a given amount of sketching space.)

The key observation we make is that, given a desired level of accuracy, the space required is proportional to the *product of the self-join sizes* of relations R_1, \dots, R_r over the join attributes (Theorem 1). Further, in practice, join-attribute domains are frequently skewed and the skew is often concentrated in different regions for different attributes. As a consequence, we can exploit approximate knowledge of the data distribution(s) to intelligently partition the domains of (some subset of) join attributes so that, for each resulting partition \mathbf{p} of the combined attribute space, the product of self-join sizes of relations restricted to \mathbf{p} is very small compared to the same product over the entire (un-partitioned) attribute space (i.e., $\prod_v \text{SJ}_v$). Thus, letting $X_{Q,\mathbf{p}}$ denote the sketch-based estimate for the portion of Q that corresponds to partition \mathbf{p} of the attribute space, we can expect the variance $\text{Var}[X_{Q,\mathbf{p}}]$ to be much smaller than $\text{Var}[X_Q]$.

Now, consider a scheme that averages over $s_{\mathbf{p}}$ iid instances of the sketching estimate $X_{Q,\mathbf{p}}$ for partition \mathbf{p} , and defines the overall estimate X_Q for query Q as the sum of these averages over all partitions \mathbf{p} . We can then show that $E[X_Q] = Q$ and $\text{Var}[X_Q] = \sum_{\mathbf{p}} \frac{\text{Var}[X_{Q,\mathbf{p}}]}{s_{\mathbf{p}}}$. Clearly, achieving small self-join sizes and variances

$\text{Var}[X_{Q,\mathbf{p}}]$ for the attribute-space partitions \mathbf{p} means that the total number of iid sketch instances $\sum_{\mathbf{p}} s_{\mathbf{p}}$ required to guarantee the prescribed accuracy level of our X_Q estimator is also small. We formalize the above intuition in the following subsection and then present our sketch-partitioning results and algorithms for both single- (Sect. 4.2) as well as multi-query (Sect. 4.3) environments.

4.1 Our General Technique

Consider once again the COUNT aggregate query Q from Sect. 2 with n equi-join constraints over relations R_1, \dots, R_r . In general, our sketch-partitioning techniques partition the domain of each join attribute $R_i.A_j$ into $m_j \geq 1$ disjoint subsets denoted by $P_{j,1}, \dots, P_{j,m_j}$. Further, the domains of a join-attribute pair $R_i.A_j$ and $R_k.A_l$ are partitioned identically (note that $\text{dom}(R_i.A_j) = \text{dom}(R_k.A_l)$). This partitioning on individual attributes induces a partitioning of the combined (multi-dimensional) join-attribute space, which we denote by \mathcal{P} . Thus, mapping attribute $R_i.A_j$ to dimension j , we get that $\mathcal{P} = \{(P_{1,l_1}, \dots, P_{n,l_n}) : 1 \leq l_j \leq m_j\}$. Each element $\mathbf{p} \in \mathcal{P}$ identifies a unique partition of the global attribute space, and we represent by $\mathbf{p}[R_i.A_j]$ the partition of attribute $R_i.A_j$ in \mathbf{p} .

For each partition $\mathbf{p} \in \mathcal{P}$, we construct random variables $X_{Q,\mathbf{p}}$ that estimate Q on attribute values in partition \mathbf{p} , in a manner similar to X_Q in Sect. 2. Thus, for each partition \mathbf{p} , we construct atomic sketches $X_{v,\mathbf{p}}$ for vertices v of the join graph $\mathcal{J}(Q)$. More specifically, for each edge $e = \langle v, w \rangle$ in $\mathcal{J}(Q)$, we have an independent family of random variables $\{\xi_l^{e,\mathbf{p}} : l \in \mathbf{p}[A_v(e)]\}$. Let e_1, \dots, e_k be the edges incident on v , and $f_v(i_1, \dots, i_k)$ be the number of tuples $t \in R(v)$ such that $t[A_v(e_j)] = i_j$, for $1 \leq j \leq k$. Then, the atomic sketch for partition \mathbf{p} at v is $X_{v,\mathbf{p}} = \sum_{i_1 \in \mathbf{p}[A_v(e_1)]} \dots \sum_{i_k \in \mathbf{p}[A_v(e_k)]} f_v(i_1, \dots, i_k) \prod_{j=1}^k \xi_j^{e_j,\mathbf{p}}$. Variable $X_{Q,\mathbf{p}}$ is then obtained as the product of $X_{v,\mathbf{p}}$'s over all the vertices, i.e., $X_{Q,\mathbf{p}} = \prod_v X_{v,\mathbf{p}}$. It is easy to verify that $E[X_{Q,\mathbf{p}}]$ is equal to the number of tuples in the join result for partition \mathbf{p} and thus, by linearity of expectation, $E[\sum_{\mathbf{p}} X_{Q,\mathbf{p}}] = \sum_{\mathbf{p}} E[X_{Q,\mathbf{p}}] = Q$.

By independence across partitions, we have $\text{Var}[\sum_{\mathbf{p}} X_{Q,\mathbf{p}}] = \sum_{\mathbf{p}} \text{Var}[X_{Q,\mathbf{p}}]$. As in [2, 9], to reduce the variance of our partitioned estimator, we construct iid instances of each $X_{Q,\mathbf{p}}$. However, since $\text{Var}[X_{Q,\mathbf{p}}]$ may differ widely across the partitions, we can obtain larger reductions in the overall variance by maintaining a larger number of copies for partitions with a higher variance. Let $s_{\mathbf{p}}$ denote the number of iid copies of the variable $X_{Q,\mathbf{p}}$ maintained for partition \mathbf{p} and let X_Q be computed as the sum of the averages of these $s_{\mathbf{p}}$ copies of $X_{Q,\mathbf{p}}$ over all the partitions. Then, since averaging over iid copies does not alter the expectation, we get that $E[X_Q] = Q$.

The success of our sketch-partitioning approach clearly hinges on being able to efficiently compute $X_{v,\mathbf{p}}$ for each (vertex, partition) pair as data tuples are streaming in. For every tuple $t \in R(v)$ in the stream and for every partition \mathbf{p} such that t lies in \mathbf{p} , we add to $X_{v,\mathbf{p}}$ the quantity $\prod_{j=1}^k \xi_t^{e_j,\mathbf{p}}$. (Note that a tuple t in $R(v)$ typically carries only a subset of the join attributes, so it can belong to multiple partitions \mathbf{p} .) Our sketch-partitioning techniques make the process of identifying

the relevant partitions for a tuple very efficient by using the (approximate) stream statistics to group contiguous regions of values in the domain of each attribute $R_i.A_j$ into a small number of coarse *buckets* (e.g., histogram statistics trivially give such a bucketization). Then, each of the m_j partitions for attribute $R_i.A_j$ comprises a subset of such buckets and each bucket stores an identifier for its corresponding partition. Since the number of such buckets is typically small, given an incoming tuple t , the bucket containing $t[R_i.A_j]$ (and, therefore, the relevant partition along $R_i.A_j$) can be determined very quickly (e.g., using binary or linear search). This allows us to very efficiently determine the relevant partitions \mathbf{p} for streaming data tuples.

The total storage required for the atomic sketches over all the partitions is $O(\sum_{\mathbf{p}} s_{\mathbf{p}} \sum_{R_i.A_j} \log |\text{dom}(R_i.A_j)|)$ to compute the final estimate X_Q for query Q .³ Our sketch-partitioning approach still needs to address two very important issues: (i) selecting a good set of partitions \mathcal{P} ; and (ii) determining the number of iid copies $s_{\mathbf{p}}$ of $X_{Q,\mathbf{p}}$ to be constructed for each partition \mathbf{p} . Clearly, effectively addressing these issues is crucial to our final goal of minimizing the overall space allocated to the sketch while guaranteeing the required accuracy ϵ for X_Q . Specifically, we aim to compute a partitioning \mathcal{P} and space allocation $s_{\mathbf{p}}$ to each partition \mathbf{p} such that the estimate X_Q has at most ϵ relative error and $\sum_{\mathbf{p} \in \mathcal{P}} s_{\mathbf{p}}$ is minimized.

Note that, by independence across partitions and the iid characteristics of individual atomic sketches, we have $\text{Var}[X_Q] = \sum_{\mathbf{p}} \frac{\text{Var}[X_{Q,\mathbf{p}}]}{s_{\mathbf{p}}}$. Given a attribute-space partitioning \mathcal{P} , the problem of choosing the optimal allocation of $s_{\mathbf{p}}$'s that minimizes the overall sketch space while guaranteeing the required level of accuracy for X_Q can be formulated as a concrete optimization problem. The following theorem describes how to compute such an optimal allocation.

Theorem 3 *Consider a partitioning \mathcal{P} of the join-attribute domains. Then, allocating space $s_{\mathbf{p}} = \frac{8\sqrt{\text{Var}[X_{Q,\mathbf{p}}]}\sum_{\mathbf{p}}\sqrt{\text{Var}[X_{Q,\mathbf{p}}]}}{\epsilon^2 Q^2}$ to each $\mathbf{p} \in \mathcal{P}$ ensures that estimate X_Q has at most ϵ relative error, and $\sum_{\mathbf{p}} s_{\mathbf{p}}$ is minimum.*

From the above theorem, it follows that, given a partitioning \mathcal{P} , the *optimal space allocation* for a given level of accuracy requires a total sketch space of: $\sum_{\mathbf{p}} s_{\mathbf{p}} = \frac{8(\sum_{\mathbf{p}} \sqrt{\text{Var}[X_{Q,\mathbf{p}}]})^2}{\epsilon^2 Q^2}$. Obviously, this means that the *optimal partitioning* \mathcal{P} with respect to minimizing the overall space requirements for our sketches is one that minimizes the sum $\sum_{\mathbf{p}} \sqrt{\text{Var}[X_{Q,\mathbf{p}}]}$. Thus, in the remainder of this section, we focus on techniques for computing such an optimal partitioning \mathcal{P} ; once \mathcal{P} has been found, we use Theorem 3 to compute the optimal space allocation for each partition. We first consider the simpler case of a single query, and subsequently address multiple join queries in Sect. 4.3.

³For the sake of simplicity, we approximate the storage overhead for each ξ family for partition \mathbf{p} by the constant $O(\sum_{R_i.A_j} \log |\text{dom}(R_i.A_j)|)$ instead of the more precise (and less pessimistic) $O(\sum_{R_i.A_j} \log |\mathbf{p}[R_i.A_j]|)$.

4.2 Sketch-Partitioning for Single Query

We begin by describing our techniques for computing an effective partitioning \mathcal{P} of the attribute space for the estimation of COUNT queries Q over single joins of the form $R_1 \bowtie_{R_1.A_1=R_2.A_2} R_2$. Since we only consider a single join-attribute pair, for notational simplicity, we drop the relation prefixes and refer to the attributes simply as A_1, A_2 . Also, we ignore the single join-edge in the join graph $\mathcal{J}(Q)$, and number the vertices corresponding to relations R_1 and R_2 in $\mathcal{J}(Q)$ by 1 and 2, respectively. Our partitioning algorithms rely on knowledge of approximate frequency statistics for attributes A_1 and A_2 . Typically, such approximate statistics are available in the form of per-attribute *histograms* that split the underlying data domain $\text{dom}(A_j)$ into a sequence of contiguous regions of values (termed *buckets*) and store some coarse aggregate statistics (e.g., number of tuples and number of distinct values) within each bucket.

Binary Sketch Partitioning

Consider the simple case of a *binary* partitioning \mathcal{P} of $\text{dom}(A_1)$ into two subsets P_1 and P_2 ; that is, $\mathcal{P} = \{P_1, P_2\}$. Let $f_k(i)$ denote the frequency of value $i \in \text{dom}(A_1)$ in relation R_k . For each vertex k (relation R_k), we associate with the (vertex, partition) pair (k, P_l) an atomic sketch $X_{k, P_l} = \sum_{i \in P_l} f_k(i) \xi_i^{P_l}$, where $l, k \in \{1, 2\}$. We can now define $X_{Q, P_l} = X_{1, P_l} X_{2, P_l}$ for $l \in \{1, 2\}$. Then, from Theorem 1, we get that the variance $\text{Var}[X_{Q, P_l}]$ is as follows:

$$\text{Var}[X_{Q, P_l}] \leq \sum_{i \in P_l} f_1(i)^2 \sum_{i \in P_l} f_2(i)^2. \quad (1)$$

Theorem 3 tells us that the overall storage is proportional to $\sqrt{\text{Var}[X_{Q, P_1}]} + \sqrt{\text{Var}[X_{Q, P_2}]}$. Thus, to minimize the total sketching space through partitioning, we need to find the partitioning $\mathcal{P} = \{P_1, P_2\}$ that minimizes $\sqrt{\text{Var}[X_{Q, P_1}]} + \sqrt{\text{Var}[X_{Q, P_2}]}$; that is, we aim to find a partitioning \mathcal{P} that minimizes the function:

$$\mathcal{F}(\mathcal{P}) = \sqrt{\sum_{i \in P_1} f_1(i)^2 \sum_{i \in P_1} f_2(i)^2} + \sqrt{\sum_{i \in P_2} f_1(i)^2 \sum_{i \in P_2} f_2(i)^2}. \quad (2)$$

Clearly, a brute-force solution to this problem is extremely inefficient as it requires $O(2^{\text{dom}(A_1)})$ time (proportional to the number of all possible partitionings of $\text{dom}(A_1)$). Fortunately, we can take advantage of the following classic theorem from the classification-tree literature [5] to design a much more efficient *optimal* algorithm.

Theorem 4 ([5]) *Let $\Phi(x)$ be a concave function of x defined on some compact domain $\hat{\mathcal{D}}$. Let $P = \{1, \dots, d\}$, $d \geq 2$, and $\forall i \in P$ let $q_i > 0$ and r_i be real*

	1	2	3	4
$f_1(i)$	20	5	10	2
$f_2(i)$	2	15	3	10

numbers with values in $\hat{\mathcal{D}}$ not all equal. Then one of the partitions $\{P_1, P_2\}$ of P that minimizes $\sum_{i \in P_1} q_i \Phi(\frac{\sum_{i \in P_1} q_i r_i}{\sum_{i \in P_1} q_i}) + \sum_{i \in P_2} q_i \Phi(\frac{\sum_{i \in P_2} q_i r_i}{\sum_{i \in P_2} q_i})$ has the property that $\forall i_1 \in P_1, \forall i_2 \in P_2, r_{i_1} < r_{i_2}$.

To see how Theorem 4 applies to our partitioning problem, let $i \in \text{dom}(A_1)$, and set $r_i = \frac{f_1(i)^2}{f_2(i)^2}$, $q_i = \frac{f_2(i)^2}{\sum_{j \in \text{dom}(A_1)} f_2(j)^2}$. Substituting in Eq. (2), we obtain

$$\begin{aligned} \mathcal{F}(\mathcal{P}) &= \sqrt{\sum_{i \in P_1} f_2(i)^2 \sum_{i \in P_1} f_2(i)^2 r_i} + \sqrt{\sum_{i \in P_2} f_2(i)^2 \sum_{i \in P_2} f_2(i)^2 r_i} \\ &= \sum_i f_2(i)^2 \left[\sqrt{\sum_{i \in P_1} q_i \sum_{i \in P_1} q_i r_i} + \sqrt{\sum_{i \in P_2} q_i \sum_{i \in P_2} q_i r_i} \right] \\ &= \sum_i f_2(i)^2 \left[\sum_{i \in P_1} q_i \sqrt{\frac{\sum_{i \in P_1} q_i r_i}{\sum_{i \in P_1} q_i}} + \sum_{i \in P_2} q_i \sqrt{\frac{\sum_{i \in P_2} q_i r_i}{\sum_{i \in P_2} q_i}} \right]. \end{aligned}$$

Except for the constant factor $\sum_{i \in \text{dom}(A_1)} f_2(i)^2$ (which is always nonzero if $R_2 \neq \emptyset$), our objective function \mathcal{F} now has exactly the form prescribed in Theorem 4 with $\Phi(x) = \sqrt{x}$. Since $f_1(i) \geq 0$, $f_2(i) \geq 0$ for $i \in \text{dom}(A_1)$, we have $r_i \geq 0$, $q_i \geq 0$, and $\forall P_l \subseteq \text{dom}(A_1)$, $\frac{\sum_{i \in P_l} q_i}{\sum_{i \in P_l} q_i r_i} \geq 0$. So, all that remains to be shown is that \sqrt{x} is concave on $\text{dom}(A_1)$. Since concaveness is equivalent to negative second derivative and $(\sqrt{x})'' = -1/4x^{-3/2} \leq 0$, Theorem 4 applies.

Applying Theorem 4 essentially reduces the search space for finding an optimal partitioning of $\text{dom}(A)$ from exponential to linear, since only partitionings in the order of increasing r_i 's need to be considered. Thus, our optimal binary-partitioning algorithm for minimizing $\mathcal{F}(\mathcal{P})$ simply orders the domain values in increasing order of frequency ratios $\frac{f_1(i)}{f_2(i)}$, and only considers partition boundaries between two consecutive values in that order; the partitioning with the smallest resulting value for $\mathcal{F}(\mathcal{P})$ gives the optimal solution.

Example 4 Consider the join $R_1 \bowtie_{A_1=A_2} R_2$ of two relations R_1 and R_2 with $\text{dom}(A_1) = \text{dom}(A_2) = \{1, 2, 3, 4\}$. Also, let the frequency $f_k(i)$ of domain values i for relations R_1 and R_2 be as follows:

Without partitioning, the number of copies s of the sketch estimator X_Q to ensure ϵ relative error is given by $s = \frac{8\text{Var}[X_Q]}{\epsilon^2 Q^2}$, where $\text{Var}[X_Q] \leq 529 \cdot 338 + 165^2 \leq 206027$ by Eq. (1). Now consider the binary partitioning \mathcal{P} of $\text{dom}(A_1)$ into

$P_1 = \{1, 3\}$ and $P_2 = \{2, 4\}$. The total number of copies $\sum_p s_p$ of the sketch estimators $X_{Q,p}$ for partitions P_1 and P_2 is $\sum_p s_p = \frac{8(\sqrt{\text{Var}[X_{Q,P_1}]} + \sqrt{\text{Var}[X_{Q,P_2}]})^2}{\epsilon^2 Q^2}$ (by Theorem 3), where $(\sqrt{\text{Var}[X_{Q,P_1}]} + \sqrt{\text{Var}[X_{Q,P_2}]})^2 \leq (\sqrt{6400} + \sqrt{6400})^2 \leq 25600$. Thus, using this binary partitioning, the sketching space requirements are reduced by a factor of $\frac{s}{\sum_p s_p} = \frac{206027}{25600} \approx 7.5$.

Note that the partitioning \mathcal{P} with $P_1 = \{1, 3\}$ and $P_2 = \{2, 4\}$ also minimizes the function $\mathcal{F}(\mathcal{P})$ defined in Eq. (2). Thus, our approximation algorithm based on Theorem 4 returns the above partitioning \mathcal{P} . Essentially, since $r_1 = \frac{20^2}{2^2} = 100$, $r_2 = \frac{5^2}{15^2} = 1/9$, $r_3 = \frac{10^2}{3^2} = 100/9$ and $r_4 = \frac{2^2}{10^2} = 1/25$, only the three split points in the sequence 4, 2, 3, 1 of domain values arranged in the increasing order of r_i need to be considered. Of the three potential split points, the one between 2 and 3 results in the smallest value (177) for $\mathcal{F}(\mathcal{P})$.

K-ary Sketch Partitioning

We now describe how to extend our earlier results to more general partitionings comprising $m \geq 2$ domain partitions. By Theorem 3, we aim to find a partitioning $\mathcal{P} = \{P_1, \dots, P_m\}$ of $\text{dom}(A_1)$ that minimizes $\sqrt{\text{Var}[X_{Q,P_1}]} + \dots + \sqrt{\text{Var}[X_{Q,P_m}]}$, where each $\text{Var}[X_{Q,P_l}]$ is computed as in Eq. (1). Once again, we substitute the variance formulas with the product of self-join sizes (Theorem 1); thus, we seek to find a partitioning $\mathcal{P} = \{P_1, \dots, P_m\}$ that minimizes the function

$$\mathcal{F}(\mathcal{P}) = \sqrt{\sum_{i \in P_1} f_1(i)^2 \sum_{i \in P_1} f_2(i)^2} + \dots + \sqrt{\sum_{i \in P_m} f_1(i)^2 \sum_{i \in P_m} f_2(i)^2}. \quad (3)$$

A brute-force solution to minimizing $\mathcal{F}(\mathcal{P})$ requires an impractical $O(m^{\text{dom}(A_1)})$ time. Fortunately, we have shown the following generalization of Theorem 4 that allows us to drastically reduce the problem search space and design a much more efficient algorithm.

Theorem 5 Consider the function $\Psi(P_1, \dots, P_m) = \sum_{l=1}^m \sum_{i \in P_l} q_i \Phi\left(\frac{\sum_{i \in P_l} q_i r_i}{\sum_{i \in P_l} q_i}\right)$, where Φ , q_i and r_i are defined as in Theorem 4 and $\{P_1, \dots, P_m\}$ is a partitioning of $P = \{1, \dots, d\}$. Then among the partitionings that minimize $\Psi(P_1, \dots, P_m)$ there is one partitioning $\{P_1, \dots, P_m\}$ with the following property π : $\forall l, l' \in \{1, \dots, m\} : l < l' \implies \forall i \in P_l \forall i' \in P_{l'} r_i < r_{i'}$.

As described in Sect. 4.2, our objective function $\mathcal{F}(\mathcal{P})$ can be expressed as $\sum_{i \in \text{dom}(A_1)} f_2(i)^2 \Psi(P_1, \dots, P_m)$, where $\Phi(x) = \sqrt{x}$, $r_i = \frac{f_1(i)^2}{f_2(i)^2}$ and $q_i = \frac{f_2(i)^2}{\sum_{j \in \text{dom}(A_1)} f_2(j)^2}$; thus, minimizing $\mathcal{F}(\{P_1, \dots, P_m\})$ is equivalent to minimizing $\Psi(P_1, \dots, P_m)$. By Theorem 5, to find the optimal partitioning for Ψ , all we have

to do is to consider an arrangement of elements i in $P = \{1, \dots, d\}$ in the order of increasing r_i 's, and find $m - 1$ split points in this sequence such that Ψ for the resulting m partitions is as small as possible. The optimum $m - 1$ split points can be efficiently found using *dynamic programming*, as follows. Without loss of generality, assume that $1, \dots, d$ is the sequence of elements in P in increasing value of r_i . For $1 \leq u \leq d$ and $1 \leq v \leq m$, let $\psi(u, v)$ be the value of Ψ for the optimal partitioning of elements $1, \dots, u$ (in order of increasing r_i) in v parts. The equations describing our dynamic-programming algorithm are:

$$\begin{aligned}\psi(u, 1) &= \sum_{i=1}^u q_i \Phi\left(\frac{\sum_{i=1}^u q_i r_i}{\sum_{i=1}^u q_i}\right), \\ \psi(u, v) &= \min_{1 \leq j < u} \left\{ \psi(j, v-1) + \sum_{i=j+1}^u q_i \Phi\left(\frac{\sum_{i=j+1}^u q_i r_i}{\sum_{i=j+1}^u q_i}\right) \right\}, \quad v > 1.\end{aligned}$$

The correctness of our algorithm is based on the linearity of Ψ . Also let $p(u, v)$ be the index of the last element in partition $v - 1$ of the optimal partitioning of $1, \dots, u$ in v parts (so that the last partition consists of elements between $p(u, v) + 1$ and u). Then, $p(u, 1) = 0$ and for $v > 1$, $p(u, v) = \arg \min_{1 \leq j < u} \{\psi(j, v - 1) + \sum_{i=j+1}^u q_i \Phi(\frac{\sum_{i=j+1}^u q_i r_i}{\sum_{i=j+1}^u q_i})\}$. The actual best partitioning can then be reconstructed from the values of $p(u, v)$ in time $O(m)$; essentially, the $(m - 1)$ th split point of the optimal partitioning is $p(d, m)$, the split point preceding it is $p(p(d, m), m - 1)$, and so on. The space complexity of the algorithm is obviously $O(md)$ and the time complexity is $O(md^2)$, since we need $O(d)$ time to find the index j that achieves the minimum for a fixed u and v , and the function $\Phi()$ for sequences of consecutive elements can be computed in time $O(d^2)$.

Sketch-Partitioning for Multi-Join Queries

When queries contain 2 or more joins, unfortunately, the problem of computing an optimal partitioning becomes intractable [9]. However, if attribute value distributions within each relation are independent, then we can show that a simple approach that combines the optimal (local) partitions for each individual join attribute yields an optimal partitioning for the (global) multi-join attribute space. Thus, under this assumption, we can use the dynamic programming algorithm described above to compute the m_j optimal partitions $P_{j,1}, \dots, P_{j,m_j}$ for each join attribute $R_i.A_j$, and $\mathcal{P} = \{(P_{1,l_1}, \dots, P_{n,l_n}) : 1 \leq l_j \leq m_j\}$ is then our desired optimal partitioning of the global attribute space.

4.3 Sketch-Partitioning for Multiple Join Queries

In the previous subsections, we described how sketch partitioning can be used to reduce sketching space requirements for a single multi-join query. We now study how we can incorporate these sketch partitioning ideas into the sketch sharing techniques (from Sect. 3) to improve space utilization and answer accuracy when processing a collection \mathcal{Q} of queries.

Suppose that we could compute a *single* optimal partitioning \mathcal{P} of the global join attribute space for queries in \mathcal{Q} using a variant our earlier proposed dynamic programming algorithm. Then, we can simply use our sketch sharing techniques to estimate the partial result $Q_{\mathbf{p}}$ for each query, partition pair (Q, \mathbf{p}) , where $Q \in \mathcal{Q}$ and $\mathbf{p} \in \mathcal{P}$. These estimates $Q_{\mathbf{p}}$ for the various partitions can then be summed up to derive the final estimate for query Q . Note that computing query estimates for a partition \mathbf{p} simply involves running our algorithms with attribute domains restricted to lie within the partition \mathbf{p} .

A major challenge here is to derive a single partitioning \mathcal{P} that would be optimal for all the queries in \mathcal{Q} . The reason this is a non-trivial problem is that a join attribute $R_i.A_j$ may be paired with different attributes in the various queries, and each query may induce a very different partitioning depending on the attributes being paired. One possibility here is to compute multiple partitionings for $R_i.A_j$ (one per query that $R_i.A_j$ appears in) and then try to massage them so that they become identical. There may be other approaches as well, and we intend to explore these further as part of future work.

5 Conclusions

In this chapter, we considered the problem of approximatively answering multiple general *aggregate* SQL queries over continuous data streams with limited memory. Our approach is based on computing small “sketch” summaries of the streams that are used to provide approximate answers of complex multi-join aggregate queries with provable approximation guarantees. When multiple queries need to be concurrently processed, *sketch sharing*—the technique we proposed to reuse the space between queries—can reduce the amount of memory required. We provided necessary and sufficient conditions for sketch sharing to result in correct estimation, and provided a greedy algorithm to solve the provably \mathcal{NP} -hard problem of optimally sharing the sketches and allocating the available space. Furthermore, since the degradation of the approximation quality due to the high variance of our randomized sketch synopses may be a concern in practical situations, we developed novel *sketch-partitioning* techniques. Our proposed methods take advantage of existing statistical information on the stream to intelligently partition the domain of the underlying attribute(s) and, thus, decompose the sketching problem in a way that provably tightens the approximation guarantees.

References

1. S. Acharya, P.B. Gibbons, V. Poosala, S. Ramaswamy, Join synopses for approximate query answering, in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania (1999), pp. 275–286
2. N. Alon, P.B. Gibbons, Y. Matias, M. Szegedy, Tracking join and self-join sizes in limited storage, in *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania (1999)
3. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, Philadelphia, Pennsylvania (1996), pp. 20–29
4. S. Babu, J. Widom, Continuous queries over data streams. *ACM SIGMOD Rec.* **30**(3), 109–120 (2001)
5. L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone, *Classification and Regression Trees* (Chapman & Hall, London, 1984)
6. K. Chakrabarti, M. Garofalakis, R. Rastogi, K. Shim, Approximate query processing using wavelets, in *Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt (2000), pp. 111–122
7. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows, in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California (2002)
8. A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi, Sketch-based multi-query processing over data streams. Manuscript available at www.cise.ufl.edu/~adobra/papers/sketch-mqo.pdf
9. A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi, Processing complex aggregate queries over data streams, in *Proc. of the 2002 ACM SIGMOD Intl. Conference on Management of Data*, Madison, Wisconsin (2002), pp. 61–72
10. P. Domingos, G. Hulten, Mining high-speed data streams, in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, Massachusetts (2000), pp. 71–80
11. M. Garofalakis, P.B. Gibbons, Approximate query processing: taming the terabytes in 27th Intl. Conf. on Very Large Data Bases, Rome, Italy (2001). Tutorial
12. J. Gehrke, F. Korn, D. Srivastava, On computing correlated aggregates over continual data streams, in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California (2001)
13. P.B. Gibbons, Y. Matias, V. Poosala, Fast incremental maintenance of approximate histograms, in *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece (1997), pp. 466–475
14. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *Proceedings of the 27th International Conference on Very Large Data Bases*, Rome, Italy (2000)
15. M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California (2001)
16. S. Guha, N. Koudas, K. Shim, Data streams and histograms, in *Proceedings of the 2001 ACM Symposium on Theory of Computing (STOC)*, Crete, Greece (2001)
17. S. Guha, N. Mishra, R. Motwani, L. O'Callaghan, Clustering data streams, in *Proceedings of the 2000 Annual Symposium on Foundations of Computer Science (FOCS)* (2000)
18. P.J. Haas, J.M. Hellerstein, Ripple joins for online aggregation, in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania (1999), pp. 287–298
19. Y.E. Ioannidis, V. Poosala, Histogram-based approximation of set-valued query answers, in *Proceedings of the 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland (1999)

20. G. Manku, S. Rajagopalan, B. Lindsay, Random sampling techniques for space efficient online computation of order statistics of large datasets, in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania (1999)
21. Y. Matias, J.S. Vitter, M. Wang, Dynamic maintenance of wavelet-based histograms, in *Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt (2000)
22. S.M. Stefanov, *Separable Programming*. Applied Optimization, vol. 53 (Kluwer Academic, Norwell, 2001)
23. J. Vitter, Random sampling with a reservoir. *ACM Trans. Math. Softw.* **11**(1), 37–57 (1985)
24. J.S. Vitter, M. Wang, Approximate computation of multidimensional aggregates of sparse data using wavelets, in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania (1999)

Approximate Histogram and Wavelet Summaries of Streaming Data

S. Muthukrishnan and Martin Strauss

1 Introduction

We study a synopsis abstract data structure similar to an array abstract data type commonly seen in textbooks. Specifically, we model an array \mathbf{A} of N real values, $\{i : 0 \leq i < N\} = [0, N]$. Like to an array, a user poses a *point query*, i.e., an index i , and expects $\mathbf{A}[i]$ in return. Compared with an array, however, the modes for updating data values are restricted. First we consider only ordered and aggregated data: the values $\mathbf{A}[0], \mathbf{A}[1], \dots$, are presented in order, followed by point queries.

After considering the basic abstract data type, we generalize in several directions:

- The user can pose *range queries*, i.e., a pair (ℓ, r) of indices for which the ideal answer is $\sum_{\ell \leq i < r} \mathbf{A}[i]$. This is more general than standard arrays that support point queries.
- The data can be updated additively. That is, interspersed with point or range queries are update commands of the form “add x to $\mathbf{A}[i]$,” where x may be positive or negative. Note that this is still less general than an array data type since an update of the form “change $\mathbf{A}[i]$ to x ,” which is supported by arrays, is not supported here.
- The data may be multidimensional. That is, in dimension d , the index is a d -tuple of integers. Ideally, multidimensional and single dimensional arrays are

S. Muthukrishnan supported by NSF ITR 0220280. M. Strauss supported in part by NSF DMS-0354600 and NSF DMS-0510203.

S. Muthukrishnan (✉)
Department of Computer Science, Rutgers University, Piscataway, NJ, USA
e-mail: muthu@cs.rutgers.edu

M. Strauss
Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA
e-mail: martinjs@umich.edu

equivalent from the perspective of point queries, but are different under range queries. Whether an array is considered to be single- or multi-dimensional also matters to the approximate implementations we introduce below.

We will study approximate versions of the abstract data structure above. The response to a point query i to instance \mathbf{A} is not $\mathbf{A}[i]$ but $\mathbf{R}[i]$, where \mathbf{R} is some approximate representation for \mathbf{A} . The error of this answer is $\mathbf{R}[i] - \mathbf{A}[i]$. We report the quality of a representation as $\sum_i |\mathbf{R}[i] - \mathbf{A}[i]|^2$, written $\|\mathbf{R} - \mathbf{A}\|^2$, where the sum is over all queries. A approximate data structure can have any internal storage format, but must lead to a vector \mathbf{R} of length N —the vector of answers that it gives.

Traditionally, the advantage of implementing the approximate abstract data type instead of the ideal abstract data type is that one can store an approximate representation in much less space than the full data set. That is, the approximate data structure is a *summary*.

We will be interested in two related types of sparse summaries, *histograms* and *Haar wavelets*.

Histograms A B -bucket *histogram* of length N is a partition of $[0, N)$ into intervals $[b_0, b_1) \cup [b_1, b_2) \cup \dots \cup [b_{B-1}, b_B)$, where $b_0 = 0$ and $b_B = N$, together with a collection of B heights h_j , for $0 \leq j < B$, one for each bucket. On point query i , the histogram answer is h_j , where j is the index of the bucket containing i ; that is, the unique j with $b_j \leq i < b_{j+1}$. In vector notation, we write χ_S for the vector that is 1 on the set S and zero elsewhere. Thus the answer vector of a histogram is $\mathbf{R} = \sum_{0 \leq j < B} h_j \chi_{[b_j, b_{j+1})}$.

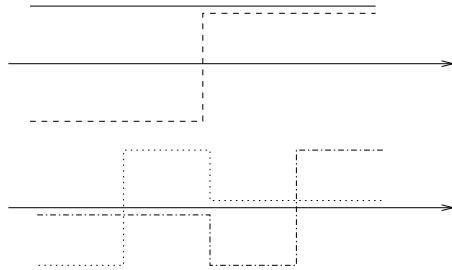
Thus, in building a B -bucket histogram, we want to choose $B - 1$ boundaries b_j and B heights h_j that tend to minimize the sum square error $\|\mathbf{A} - \sum_{0 \leq j < B} h_j \times \chi_{[b_j, b_{j+1})}\|^2$. Once we have chosen the boundaries, the best bucket height on an interval I is the average of \mathbf{A} over I . Put another way, let $\psi = \frac{1}{\sqrt{|I|}} \chi_I$. Then $\|\psi\| = 1$ and the vector resulting in the best histogram that is a multiple of ψ is $\langle \mathbf{A}, \psi \rangle \psi$, where $\langle x, y \rangle$ denotes the dot product.

Haar Wavelets We now define Haar wavelets concisely. First, we assume that N is a power of 2. This restriction is easily overcome in our algorithms.

A useful term is the *support* of a vector \mathbf{v} . The support of \mathbf{v} denoted $\text{supp}(\mathbf{v})$ is $\{i : \mathbf{v}[i] \neq 0\}$. A *dyadic* subinterval of $[0, N)$ is, intuitively, either $[0, N)$ itself or, recursively, a dyadic subinterval of $[0, N/2)$ or of $[N/2, N)$. Formally, it is an interval of the form $[k2^j, (k+1)2^j)$, where j and k must be *integers*.

A *Haar wavelet* vector ψ of length N is either the vector $\frac{1}{\sqrt{N}} \chi_{[0, N)}$ or $\frac{-1}{\sqrt{|L \cup R|}} \chi_L + \frac{1}{\sqrt{|L \cup R|}} \chi_R$, where L and R are the left and right halves of a dyadic interval, respectively. (Note that $\frac{1}{\sqrt{N}} \chi_{[0, N]}$ is *anomalous*; it doesn't fit the pattern of all other Haar wavelet vectors.) We typically drop the term "Haar." There are N wavelet vectors in all and they form an *orthonormal basis*, i.e., if ψ and φ are wavelet vectors, then $\langle \psi, \varphi \rangle$ is 1 if $\psi = \varphi$ and $\langle \psi, \varphi \rangle$ is zero, otherwise. The four coarsest Haar wavelets (i.e., those with largest support) are pictured in Fig. 1.

Fig. 1 The four coarsest Haar wavelets with normalization modified for visibility



A wavelet term is the index j of a wavelet vector ψ_j together with a (wavelet) coefficient, c_j . A B -term wavelet representation is a collection of B wavelet terms, whose answer vector is $\sum_{j \in \Lambda} c_j \psi_j$, where $|\Lambda| = B$. In building a B -term wavelet representation, we want to choose a set Λ of B wavelet indices and coefficients c_j for $j \in \Lambda$ that tends to minimize the sum square error $\|\mathbf{A} - \sum_{j \in \Lambda} c_j \psi_j\|^2$.

Note that any signal is exactly represented by an N -bucket histogram, where each bucket has size 1. Similarly, any signal is exactly represented by an N -term wavelet representation. For signal \mathbf{A} , define $c_j = \langle \mathbf{A}, \psi_j \rangle$. Then one can show that $\mathbf{A} = \sum_j c_j \psi_j$, where the sum is over all the wavelet indexes. We therefore refer to c_j as the j th wavelet coefficient of \mathbf{A} . One can also show (Parseval's equality) that $\|\mathbf{A}\|^2 = \sum_j c_j^2$. It follows that, for any set Λ of wavelet indices, $\|\mathbf{A}\|^2 = \|\mathbf{A} - \sum_{j \in \Lambda} c_j \psi_j\|^2 + \sum_j |c_j|^2$. Since $\|\mathbf{A}^2\|$ is constant, our goal of minimizing $\|\mathbf{A} - \sum_{j \in \Lambda} c_j \psi_j\|^2$ is achieved by choosing Λ to maximize $\sum_j |c_j|^2$, provided the c_j 's are optimally chosen to be the corresponding wavelet coefficients of \mathbf{A} . Similarly, if $\psi = \frac{1}{\sqrt{|I|}} \chi_I$ and $c = \langle \mathbf{A}, \psi \rangle$, then the error $\|\mathbf{A} - c\psi\|^2$ is $\|\mathbf{A}\|^2 - c^2$.

The problem we address is as follows:

Problem Fix signal \mathbf{A} of length N . Fix integer B and positive reals ϵ and δ . Find a B -bucket histogram (or B -term wavelet representation, respectively) \mathbf{R} such that, except with probability δ ,

$$\|\mathbf{A} - \mathbf{R}\|^2 \leq (1 + \epsilon) \|\mathbf{A} - \mathbf{R}_{\text{opt}}\|^2,$$

where \mathbf{R}_{opt} is the optimal B -bucket histogram (or B -term wavelet representation, respectively).

Occasionally we will also discuss approximation in the size of the summary. That is, we will produce a histogram with $B' > B$ buckets and compare its error with the best B -bucket histogram. Here we will give an upper bound for B' . If B' is guaranteed to be at most αB , then we denote the overall guarantee as a $(\alpha, 1 + \epsilon)$ -approximation. A similar approach can be used for wavelet representations, too.

The resources we use will depend quantitatively on N , B , ϵ , δ , and M , where M is a bound on the largest data item in \mathbf{A} , which we assume are integers (generally,

they are integer multiples of some smallest resolvable quantity). Several different types of resources can be measured, depending on the nature of the data.

We will consider two different ways that data may be presented to our algorithm.

Ordered Aggregate Data

The data are presented in order, as $\mathbf{A}[0], \mathbf{A}[1], \mathbf{A}[2], \dots, \mathbf{A}[N - 1]$. After the last data item is presented, the algorithm outputs a histogram or wavelet representation and halts. Here the goal is to bound the total time and total space of the algorithm. We present a deterministic construction that, for constants c_1 and c_2 , consumes total time bounded by $c_1 N + (B \log(N) \log(M)/\epsilon)^{c_2}$. For typical values of the parameters, the first term dominates; that is, the algorithm takes time linear in N , independent of (small changes in) the other parameters. The space is $(B \log(N) \log(M)/\epsilon)^{c_2}$, which is much less than N for typical settings of the parameters.

Dynamic Data

This is motivated by streaming data. Here we process an arbitrary sequence of updates of the form “add x to $\mathbf{A}[i]$ ” and “rebuild summary.” The time to rebuild the histogram and the total space are both at most proportional to $(B \log(N) \log(M) \log(1/\delta)/\epsilon)^{c_2}$. In many applications, the time to process an update must be much less than the time to build a histogram. Indeed there are algorithms that make update processing much faster without significant effect on the time to build a histogram and the total space. In this article, however, we give a construction that meets only the same general time bound of $(B \log(N) \log(M) \log(1/\delta)/\epsilon)^{c_2}$ as is satisfied for processing updates.

In what follows, we present techniques for various summaries. The techniques presented earlier will not only be useful in their own right for building summaries, but also be used as building blocks for more advanced techniques that are described later.

2 Wavelet Summaries for Ordered Aggregate Data

We are given the data in an ordered aggregate stream,

$$\mathbf{A}[0], \mathbf{A}[1], \mathbf{A}[2], \dots, \mathbf{A}[N - 1].$$

Our algorithm will consist of several stages. The first stage takes in the input stream and outputs a stream of all N wavelet terms. The second stage takes a stream of all wavelet terms and outputs just the B terms with largest coefficients.

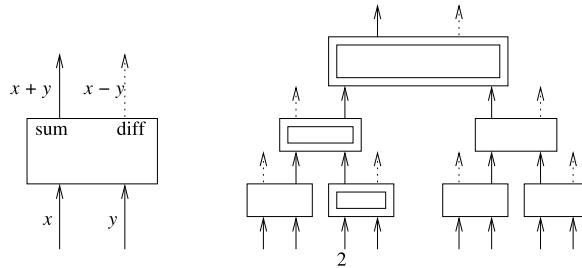


Fig. 2 Computation of all Haar wavelet coefficients by a binary tree (right) of basic filters (left), each of which computes the sum and difference of its inputs. When the second item (counting from zero) is read, only the $O(\log(N))$ basic filters in the path of 2, indicated by a double box, need to be instantiated

2.1 Online Wavelet Transform

Our goal is to consume a stream of data and output a stream of all wavelet coefficients.

A basic *filter* reads in two elements at a time. It then outputs the sum and the difference of these two elements.

Now, form an overall filter Φ by cascading combining $N - 1$ basic filters in a binary tree. The sum output of each basic filter is the input of another basic filter and the difference output of each basic filter is sent, after normalization, to an overall output stream of Φ . The sum output of the basic filter at the root of the tree also contributes to the output stream of Φ (see Fig. 2).

Each basic filter does $O(1)$ work per input and uses $O(1)$ space. There are $N - 1$ basic filters in Φ , so the total time used by Φ is $O(N)$. Finally, note that only $O(\log(N))$ basic filters need to be instantiated at any time, so Φ can be implemented using space $O(\log(N))$; see Fig. 2. It is straightforward to check that Φ computes exactly the set of wavelet coefficients. (In fact, the output of Φ is often taken as the definition of the wavelet coefficients, particularly for wavelets more general than Haar wavelets, which are defined solely in terms of the two or more outputs of the basic filters.)

2.2 Finding the Largest Terms

Given a sequence of wavelet terms, we want to keep just the B terms having largest coefficient absolute values. This can be done as follows. Create a buffer of size $4B$, and read the first B items into it. Now we maintain the invariant that our buffer has between B and $3B$ items, including the B largest items seen so far. We proceed as follows. Fill the buffer, by reading a number of items between B and $3B$. Next, ensure that the items all have unequal rank, e.g., by ordering them primarily by coefficient size and secondarily by position in the buffer. Then find an approximate

median—an item whose rank is between the 25th and 75th percentile. (This can be done deterministically in time $O(B)$ (see [2]), or probabilistically by choosing a few random elements and testing them against the entire buffer.) Finally, discard buffer elements smaller than the approximate median, restoring the invariant. The total time spent on this operation is $O(B)$ and it consumes $\Omega(B)$ items, so the time cost is $O(1)$ per item, as desired.

2.3 Analysis

It is clear by construction that the overall algorithm meets our claims. In fact, a more careful analysis shows that the worst-case work upon reading any item is at most $O(\log(N))$ for the first phase and $O(B)$ for the second phase. By using buffers of size $O(B + \log(N))$, one can arrange that the worst-case time per item is at most $O(1)$. This entire algorithm first appeared in [5] and is also discussed and used in [7].

3 Offline Histogram Algorithms

In this section, we show how to find optimal histograms in full space. We consider histograms in one dimension that are optimal for point queries.

3.1 A Quadratic Time Exact Algorithm

The data is \mathbf{A} . First, we build a structure for \mathbf{A} from which, given ℓ and r , we can get, in time $O(1)$, the height and error of an optimal 1-bucket histogram for \mathbf{A} on the interval $[\ell, r]$. This can be done in time (and space) $O(N)$ as follows: Stream over the data and build the *prefix arrays* $\mathcal{P}(\mathbf{A})$ of \mathbf{A} and $\mathcal{P}(\mathbf{A}^2)$ of \mathbf{A}^2 , where $\mathcal{P}(\mathbf{A})[j] = \sum_{0 \leq i < j} \mathbf{A}[i]$ and $\mathcal{P}(\mathbf{A}^2)[j] = \sum_{0 \leq i < j} \mathbf{A}^2[i]$, for $0 \leq j \leq N$. The optimal height on interval $[\ell, r]$ is $h = \frac{1}{r-\ell} \sum_{\ell \leq i < r} \mathbf{A}[i] = \frac{1}{r-\ell} [(\sum_{i < r} \mathbf{A}[i]) - (\sum_{i < \ell} \mathbf{A}[i])]$ and the error is $\sum_{\ell \leq i < r} (\mathbf{A}[i] - h)^2 = (\sum_{\ell \leq i < r} \mathbf{A}[i]^2) - 2h(\sum_{\ell \leq i < r} \mathbf{A}[i]) + (r - \ell)h^2$. Thus both the height and error can be computed from ℓ, r , and our structure in time $O(1)$, as claimed.

The algorithm uses dynamic programming. Fix $m' < N$ and $\ell \leq B$. Assume that, for each $m < m'$, we have found the optimal $(\ell - 1)$ -bucket histogram on the prefix of data indexed by $[0, i]$. To find the best ℓ -bucket histogram on $[0, m']$, we try all $m < m'$ for the final boundary, and form a histogram by joining the best $(\ell - 1)$ -bucket histogram on $[0, m]$ with the 1-bucket histogram on $[m, m']$. The time for this step is therefore at most $O(N)$, to try all values of m . Since this has to be repeated for all $m' < N$ and all $\ell \leq B$, the total time is $O(N^2B)$. The space is $O(BN)$, to store a table of a B -bucket histogram for each i .

This procedure works because we can decompose the error on an overall histogram into contributions from histogram fragments, in which the error attributed to a fragment depends only on the data values local to that fragment. One can substitute other error measures for sum square error in the overall algorithm. This dynamic program explicitly appears in [4].

3.2 A Linear Time Approximation Algorithm

If $B \ll N$, one can get a much faster approximate result. Instead of asking, “for each ℓ and m , what is the error k of the best ℓ -bucket histogram on $[0, m)$?” we ask, “for each ℓ and each k , what is the largest m such that there is an ℓ -bucket histogram on $[0, m)$ with error approximately k ?” Computationally, this formulation is better because we can find m by binary search in $\log(N)$ iterations instead of exhaustively in N iterations. We will need to define more precisely what is meant by “error approximately k ” and we will have to show that we can restrict the allowable values for k to a small set (of size $O(B/\epsilon)$).

As above, we first construct prefix arrays for \mathbf{A} and \mathbf{A}^2 . Construction takes time $O(N)$ and allows us to find, in constant time, the best height and error of a 1-bucket histogram over any queried subinterval.

If E_{opt} denotes the error of an optimal histogram, assume that we know an E with $\frac{1}{2}E_{\text{opt}} \leq E < E_{\text{opt}}$. Note that, if the data values are integers and $\|\mathbf{A}\|^2 \leq M$, then the error is at least $\Omega(1)$ and at most M , and there are only $O(\log(M))$ possible powers of 2 in this range. We can find the right power of 2 for E in time multiplied by $\log \log(M)$ by binary search.

Let $\delta = \Theta(\epsilon E/B)$ be a unit of error. Let Δ be an integer around $4E/\delta$, so that $2E_{\text{opt}} \leq \Delta\delta < 4E_{\text{opt}}$ and so any (partial) histogram with error $\Delta\delta$ can be discarded as not nearly optimal. We will restrict errors to be non-negative integer multiples of δ up to $\Delta\delta$.

Fix $\ell \leq B$ and integer $k \leq \Delta$. Suppose for each integer $i \leq \Delta$ we know an $m = M_{i,\ell-1}$ such that there is an $(\ell - 1)$ -bucket histogram on $[0, m)$ with error at most $(i + \ell - 1)\delta$ and no histogram on $[0, m + 1)$ with error less than $i\delta$. (Here we deem the best histogram on $[0, N + 1)$ to have error $+\infty$.) We want to find an $m' = M_{k,\ell}$ such that there is an ℓ -bucket histogram on $[0, m')$ with error at most $(k + \ell)\delta$ and no ℓ -bucket histogram on $[0, m' + 1)$ with error at most $k\delta$. To do this, try all possible values of i , put $m = M_{i,\ell-1}$, and find, by binary search, the largest m' such that the one-bucket histogram on $[m, m')$ has error at most $(k - i + 1)\delta$ on $[m, m')$. This combined with the best $(\ell - 1)$ -bucket histogram on $[0, m)$ with error at most $(i + \ell - 1)\delta$ is an ℓ -bucket histogram on $[0, m')$ with error at most $(k + \ell)\delta$, as desired. We can also compute the histograms themselves as we compute the $M_{k,\ell}$'s.

On the other hand, we need to show that there is no ℓ -bucket histogram on $[0, m' + 1)$ with error at most $k\delta$. So suppose that the optimal ℓ -bucket histogram on $[0, m' + 1)$ has error η . This histogram has some penultimate boundary, m^* , and there is some error E^* on $[0, m^*)$. Find integer i^* such that $(i^* - 1)\delta \leq E^* < i^*\delta$.

Then the error on $[m^*, m' + 1]$ is the error on $[0, m' + 1]$ less the error on $[0, m^*]$, which is at most $\eta - (i^* - 1)\delta$. Our dynamic programming algorithm will have considered $(\ell - 1)$ -bucket histograms on $[0, M_{i^*, \ell-1}]$. By definition of $M_{i^*, \ell-1}$, there is no $(\ell - 1)$ -bucket histogram on $[0, M_{i^*, \ell-1} + 1]$ with error at most $i^*\delta$. Since there is an $(\ell - 1)$ -bucket histogram on $[0, m^*]$ with error at most $i^*\delta$, it follows that $M_{i^*, \ell-1} \geq m^*$. If our algorithm attempted to extend the $(\ell - 1)$ -bucket histogram on $[0, M_{i^*, \ell-1}]$ by the bucket $[M_{i^*, \ell-1}, m' + 1] \subseteq [m^*, m' + 1]$, the error of the new bucket would be at most $\eta - (i^* - 1)\delta$. Since the algorithm rejected $m' + 1$ in favor of m' , we must have $\eta - (i^* - 1)\delta > (k - i^* + 1)\delta$, or $\eta > k\delta$, as desired.

Finally, when we have computed all $M_{i, \ell}$, we can find the least i such that $M_{i, B} < N$. Then there is no B -bucket histogram on $[0, N] \supseteq [0, M_{i, B} + 1]$ with error $i\delta$ (i.e., $E_{\text{opt}} > i\delta$) but, since $M_{i+1, B} = N$, we have found a B -bucket histogram on $[0, N]$ with error at most $((i + 1) + B)\delta \leq E_{\text{opt}} + (B + 1)\delta$. If we make $\delta = \frac{\epsilon}{B+1}E \leq \frac{\epsilon}{B+1}E_{\text{opt}}$, then our error will be at most $(1 + \epsilon)E_{\text{opt}}$.

Note that, other than producing the prefix arrays $\mathcal{P}(\mathbf{A})$ and $\mathcal{P}(\mathbf{A}^2)$, the algorithm runs in time polynomial in $\log \log(M)$, B , $\log(N)$, and $1/\epsilon$. Similarly, other than producing the prefix arrays, the algorithm requires space polynomial in B and ϵ to store a B -bucket histogram for each of $O(B/\ell)$ possible k 's. That is, this algorithm meets our desired time bound of $c_1N + (B \log(N) \log(M)/\epsilon)^{c_2}$ but not our desired space bound for dynamic data situations.

The algorithm above is somewhat reminiscent of those in [8, 9] where the authors use it in the ordered aggregate mode.

4 Basic Histograms from Wavelet Representations

In this section, we discuss several relationships between histograms and wavelet representations. We give a $(O(\log(N)), 1)$ -approximation algorithm, a $(9, 1)$ -approximation algorithm, and a $(1, 1 + \epsilon)$ -approximation algorithm.

4.1 Wavelet Representations as Histograms

First note that any histogram element χ_I on interval I can be expressed as a $2\log(N)$ -term wavelet representation. This is because $\langle \chi_I, \psi \rangle = 0$ unless $\text{supp}(\psi)$ intersects an endpoint of I . Each endpoint of I is in the support of $\log(N) + 1$ wavelet vectors, but the two wavelet vectors with support $[0, N]$ are double counted, so only $2\log(N)$ terms remain. (See Fig. 3.)

On the other hand, any wavelet term is itself a 4-bucket histogram (i.e., a histogram with 3 boundaries), so a B -term wavelet representation can be viewed as a $(3B + 1)$ -bucket histogram.

It follows that the best $2\log(N)B$ -term wavelet representation \mathbf{R} is at least as good as the best B -bucket histogram. Also, \mathbf{R} can be regarded as a $(6\log(N) + 1)$ -bucket histogram. It follows that we can find a $(O(\log(N)), 1)$ -approximation to



Fig. 3 The dot product between a wavelet and a histogram is zero unless the support of the wavelet intersects a boundary of the histogram

the best B -bucket histogram in linear time using previously discussed algorithm for computing the wavelet summary.

4.2 Wavelet Representations as Intermediate Summaries

Next, we consider the following algorithm: Given \mathbf{A} , let \mathbf{R} be the best $2 \log(N)B$ -term wavelet representation to \mathbf{A} , as above. Thus, as above, $\|\mathbf{A} - \mathbf{R}\|^2 \leq \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$, where \mathbf{H}_{opt} is the optimal B -bucket histogram for \mathbf{A} . Next, let \mathbf{H} be the best B -bucket histogram to \mathbf{R} . How good is \mathbf{H} as a summary of \mathbf{A} ?

By the triangle inequality and by optimality of \mathbf{H} for \mathbf{R} , we have

$$\begin{aligned} \|\mathbf{A} - \mathbf{H}\| &\leq \|\mathbf{A} - \mathbf{R}\| + \|\mathbf{R} - \mathbf{H}\| \\ &\leq \|\mathbf{A} - \mathbf{R}\| + \|\mathbf{R} - \mathbf{H}_{\text{opt}}\| \\ &\leq \|\mathbf{A} - \mathbf{R}\| + \|\mathbf{R} - \mathbf{A}\| + \|\mathbf{A} - \mathbf{H}_{\text{opt}}\| \\ &\leq 3\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|, \end{aligned}$$

whence $\|\mathbf{A} - \mathbf{H}\|^2 \leq 9\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$. Thus we have a $(1, 9)$ -approximation algorithm to the best B -bucket histogram.

Now consider the cost to build \mathbf{H} . It is easy to check that the boundaries of \mathbf{H} can be chosen from among the boundaries of \mathbf{R} ; this is proved in [7]. One can modify the algorithm of Sect. 3.2 by building, in time linear in $B \log(N)$, a structure that will answer, in time $O(1)$, the value $\sum_{\ell \leq i < r} \mathbf{R}[i]^2$ for ℓ and r among the boundaries of \mathbf{H} .

4.3 Robust Representations

In Sect. 4.2, we used the triangle inequality to bound $\|\mathbf{A} - \mathbf{H}\|$ by $\|\mathbf{A} - \mathbf{R}\| + \|\mathbf{R} - \mathbf{H}\|$. This is not tight in the sense that $\|\mathbf{A} - \mathbf{R}\| + \|\mathbf{R} - \mathbf{H}\|$ might actually be as big as $3\|\mathbf{A} - \mathbf{H}\|$. In this section, we will define and construct a *robust* representation \mathbf{R}_{rob} with the property that

$$\begin{cases} \|\mathbf{A} - \mathbf{H}\|^2 \approx \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2 + \|\mathbf{R}_{\text{rob}}^2 - \mathbf{H}\|^2, \\ \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2 \approx \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2 + \|\mathbf{R}_{\text{rob}}^2 - \mathbf{H}_{\text{opt}}\|^2, \end{cases}$$

where, as above, \mathbf{H} is optimal for \mathbf{R}_{rob} and \mathbf{H}_{opt} is optimal for \mathbf{A} . This will allow us to compare $\|\mathbf{A} - \mathbf{H}\|$ with $\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|$, as above, but without giving up anything to the triangle inequality. (Instead, we give up a lot less in the approximation to the Pythagorean equality.)

The following theorem relates a wavelet representation to a histogram in a more subtle manner.

Theorem 1 *For some $T \leq O(1/(\epsilon^2 \log(1/\epsilon)))$ and any signal \mathbf{A} , let \mathbf{R}_{rob} be the representation formed by greedily taking $2B \log(N)$ wavelet terms at a time until either*

- *we get $2TB \log(N)$ terms, or*
- *taking an additional $B \log(N)$ wavelet terms improves (reduces) the residual $\|\mathbf{A} - \mathbf{R}_{\text{rob}}\|_2^2$ by a factor no better than $(1 - \epsilon')$, where $\epsilon' \geq \Omega(\epsilon^2)$.*

Let \mathbf{H} be the best B -bucket representation to \mathbf{R}_{rob} . Then \mathbf{H} is a $(1, 1 + \epsilon)$ -approximation to the best B -bucket histogram to \mathbf{A} .

Proof Suppose \mathbf{R}_{rob} has $2TB \log(N)$ terms. Then

$$\|\mathbf{A} - \mathbf{R}_{\text{rob}}\|_2^2 \leq (1 - \epsilon')^{(T-1)} \|\mathbf{A} - \mathbf{R}^{(B \log(N))}\|_2^2,$$

where $\mathbf{R}^{(B \log(N))}$ is the best representation of $B \log(N)$ terms. By choice of $T = 1 + (1/\epsilon') \log(25/\epsilon'^2)$, we have $(1 - \epsilon')^{(T-1)} \approx \epsilon^2/25$ and, by previous observations, $\|\mathbf{A} - \mathbf{R}^{(B \log(N))}\| \leq \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|$, and

$$\begin{aligned} \|\mathbf{A} - \mathbf{H}\|_2 &\leq \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_2 + 2\|\mathbf{A} - \mathbf{R}_{\text{rob}}\|_2 \\ &\leq \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_2 + 2(1 - \epsilon')^{(T-1)/2} \|\mathbf{A} - \mathbf{R}^{(B \log(N))}\|_2 \\ &\leq (1 + 2\epsilon/5) \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|, \end{aligned}$$

so that $\|\mathbf{A} - \mathbf{H}\|_2^2 \leq (1 + \epsilon) \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_2^2$ for sufficiently small ϵ .

Now suppose \mathbf{R}_{rob} has fewer than $2TB \log(N)$ terms, so that \mathbf{R}_{rob} together with any additional $2B \log(N)$ terms has error at least $(1 - \epsilon') \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|_2^2$. It follows that the $2B \log(N)$ terms in \mathbf{H} or \mathbf{H}_{opt} do not improve the square error of \mathbf{R}_{rob} by more than the factor $(1 - \epsilon')$. Let $\widehat{\mathbf{H}}$, regarded as a histogram or wavelet representation, be the best linear combination of \mathbf{R}_{rob} and \mathbf{H} ; our hypothesis is that $\|\mathbf{A} - \widehat{\mathbf{H}}\|^2 \geq (1 - \epsilon') \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2 = \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2 - \epsilon' \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2$. Note that, since $\widehat{\mathbf{H}}$ is the best representation on a specified line (containing \mathbf{R}_{rob} and \mathbf{H}), it follows that there's a right angle at $\widehat{\mathbf{H}}$, so that, as in Fig. 4, we have

$$\begin{aligned} \|\mathbf{R}_{\text{rob}} - \widehat{\mathbf{H}}\|^2 &= \|\mathbf{R}_{\text{rob}} - \mathbf{A}\|^2 - \|\mathbf{A} - \widehat{\mathbf{H}}\|^2 \\ &\leq \epsilon' \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2 \\ &\leq (\epsilon^2/9) \|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2. \end{aligned}$$

That is, \mathbf{R}_{rob} is much closer to $\widehat{\mathbf{H}}$ than either \mathbf{R}_{rob} or $\widehat{\mathbf{H}}$ is to \mathbf{A} . Then, as in Fig. 4, angle $\mathbf{A}-\widehat{\mathbf{H}}-\mathbf{H}$ and angle $\mathbf{A}-\widehat{\mathbf{H}}-\mathbf{H}_{\text{opt}}$ are right angles and so $\mathbf{A}-\mathbf{R}_{\text{rob}}-\mathbf{H}$ and angle

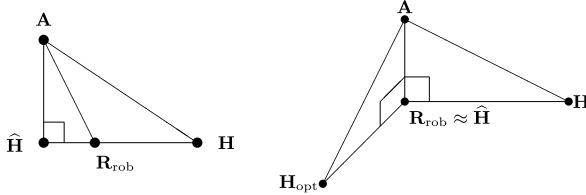


Fig. 4 Illustration of histograms in Theorem 1. On the *left*, by optimality of $\hat{\mathbf{H}}$, there is a right angle as indicated. On the *right*, since $\mathbf{R}_{\text{rob}} \approx \hat{\mathbf{H}}$, there are two near right angles at \mathbf{R}_{rob} . Since \mathbf{H} is no farther from \mathbf{R}_{rob} than \mathbf{H}_{opt} is, we conclude \mathbf{H} is almost as close to \mathbf{A} as \mathbf{H}_{opt} is. In fact, because the angles at \mathbf{R}_{rob} are not quite right angles, we can only conclude that $\|\mathbf{A} - \mathbf{H}\| \leq (1 + \epsilon)\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|$

A-R_{rob}-H_{opt} are near right angles. By comparing triangles, noting that \mathbf{H} is no farther from \mathbf{R}_{rob} than \mathbf{H}_{opt} is, we conclude \mathbf{H} is almost as close to \mathbf{A} as \mathbf{H}_{opt} is. That is, $\|\mathbf{A} - \mathbf{H}\| \leq (1 + \epsilon)\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|$. \square

In the ordered aggregate data model, the properties we have proved above immediately lead to an algorithm. We analyze the cost of constructing \mathbf{R}_{rob} . While making a single pass over the ordered aggregate data, in parallel, we find the biggest $2TB \log(N)$ terms and we compute $\|\mathbf{A}\|^2$. Both tasks can be done in time $O(N)$ and space $O(TB \log(N))$. Next we build \mathbf{R}_{rob} from our collection of $2TB \log(N)$ wavelet terms in the straightforward way. We will need to track $\|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2$ as \mathbf{R}_{rob} changes, but this can be done using the formula $\|\mathbf{A} - \mathbf{R}_{\text{rob}}\|^2 = \|\mathbf{A}^2\| - \|\mathbf{R}_{\text{rob}}\|^2$. Thus we obtain $1 + \epsilon$ approximation to the histogram in the ordered aggregate model with space as desired. The cost to produce a histogram from a wavelet representation is polynomial in B , $\log(N)$, and ϵ . This result appears in [7].

5 Histograms and Wavelets with Dynamic Data

In this section, we briefly consider data that is subject to dynamic update. We assume that we have a synopsis data structure, called an “ L^2 count sketch,” parametrized by N and η , that efficiently supports the following operations on a(n unordered) set S of size N :

- UPDATES of the form “add x to a count $\mathbf{C}[i]$ for item $i \in S$ ”;
- Unparametrized “FIND” queries. The answer is a list containing all j where $\mathbf{C}[j]^2 \geq \eta \sum_i \mathbf{C}[i]^2$;
- “COUNT” queries about item i , for which the answer is $\tilde{\mathbf{C}}[i]$ such that $|\tilde{\mathbf{C}}[i] - \mathbf{C}[i]|^2 \leq \eta \|\mathbf{C}\|^2$.

There are randomized implementations of L^2 count sketches [3, 6] that need space $(\log(N)/\eta)^{O(1)}$ and time $(\log(N)/\eta)^{O(1)}$ for either query. In particular, a FIND query can only return $(\log(N)/\eta)^{O(1)}$ values. (There is also mild but necessary dependence on the range M of counts and on the failure probability of this randomized object. We do not discuss that detail further.)

Given access to an L^2 count sketch data structure with parameter η that depends polynomially on $B \log(N)/\epsilon$, we now show how to modify the previous algorithms to produce near-optimal histogram and wavelet representations.

Recall that a signal is equivalently specified by its full set of $O(N)$ wavelet coefficients. Given input of the form “add x to $\mathbf{A}[i]$,” we form $O(\log(N))$ additive updates to the wavelet coefficients of \mathbf{A} . We then use the L^2 count sketch on the wavelet coefficients. That is how we process updates.

To build a summary in our algorithms above, we need to find the top few wavelet terms. Using the L^2 count sketch, we proceed as follows:

- Do a FIND query to find one or more wavelet index j with potentially large coefficient.
- Do COUNT queries to get an approximation \tilde{c}_j to the coefficient(s).
- Do an update of $-c_j \psi_j$; that is subtract c_j from the count of wavelet index j .

In general, we need to repeat this process several times to get enough wavelet terms ($O(TB \log(N))$) iterations will certainly suffice, though fewer iterations will also work). If we find a coefficient approximation \tilde{c}_j whose square is relatively large, after we subtract it off, other coefficients (as well as the residual coefficient $c_j - \tilde{c}_j$) will appear relatively larger—and, therefore, FIND and COUNT will be more accurate—since the sum of squares of coefficients has shrunk.

Using just the L^2 count sketch, we cannot get exact values for coefficients and therefore we cannot compare terms whose coefficients are close in magnitude, and we cannot choose the indices with largest coefficients. Nevertheless, one can show that the error in coefficients is dominated by the unavoidable error $\|\mathbf{A} - \mathbf{R}_{\text{opt}}\|$ and we will only choose a suboptimal index j if it displaces another index j' whose coefficient, $c_{j'}$, is only marginally better than c_j .

It follows that we can still build an \mathbf{R}_{rob} with the property that

- taking an additional $O(B \log(N))$ wavelet terms and changing all the coefficients to their ideal values reduces the residual $\|\mathbf{A} - \mathbf{R}_{\text{rob}}\|_2^2$ by a factor no better than $(1 - \epsilon')$, where $\epsilon' \geq \Omega(\epsilon^2)$.

This property suffices for the rest of the constructions—we can either take the top B terms for a wavelet representation or take the best B -term histogram to \mathbf{R}_{rob} . The cost in space, update time, and query time for building a representation this way is as claimed. This follows from the cost guarantees assumed about the L^2 count sketch. What we have outlined is the procedure in [6].

6 Generalized Histograms

In this section, we discuss generalizing histograms slightly. First, we consider representations that are piecewise *linear* instead of piecewise *constant*. Next, we show how to find piecewise constant histograms that are optimal for *range* queries (defined below) instead of point queries. Finally, we will discuss *multidimensional* histograms.

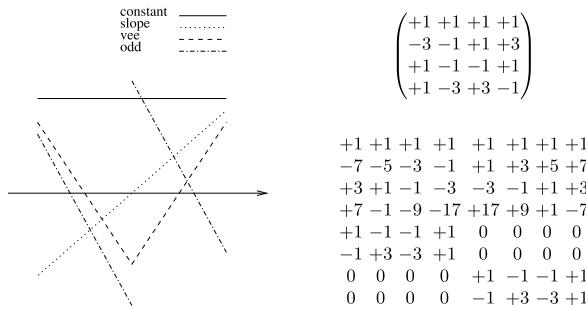


Fig. 5 Linear multiwavelets. The four coarsest linear multiwavelets are graphed, the basic four-by-four filter is given as a matrix, and the discrete versions of the coarsest eight linear multiwavelets are tabulated as rows. It is easy to see from symmetry that the linear multiwavelets form an orthonormal basis after normalization

6.1 Piecewise-Linear Representations

A B -piece piecewise-linear representation of a vector on $[0, N)$ consists of a partition of $[0, N)$ into B intervals and, for each interval, a height h_j and a slope s_j . Thus a histogram is a piecewise-linear representation in which all slopes are zero. The answer vector associated with a piecewise-linear representation is as follows. To answer a query at position i , find the bucket j such that $b_j \leq i < b_{j+1}$. Let $m_j = (b_j + b_{j+1} - 1)/2$, an integer or half-integer, be the midpoint of the bucket. Return the value $h_j + s_j(i - m_j)$. (For example, the midpoint of the 4-point bucket $[0, 4) = \{0, 1, 2, 3\}$ is 1.5.)

To build a near-optimal piecewise-linear representation, for ordered aggregate or dynamic data, we follow the above outline but replace “Haar wavelets” with “linear multiwavelets,” that we define next, based on [13]. Like the set of Haar wavelet vectors of length N , the linear multiwavelet vectors form an orthonormal family of N vectors. They consist of the appropriate scaling of the following functions. Prototypical linear multiwavelets are graphed in Fig. 5.

- An anomalous linear multiwavelet vector $\chi_{[0, N)}$. (This is called a *constant*. The constant is also the one anomalous Haar wavelet vector.)
- An anomalous linear multiwavelet vector that takes the value $i - (N - 1)/2$ at position i . (That is, the positive-slope normalized linear function of i that is orthogonal to $\chi_{[0, N)}$. This is called a *slope*.)
- On each dyadic interval $I = L \cup R$ of size at least 4 consisting of left half L and right half R with midpoint m , there is a linear multiwavelet that is a linear function on each of L and R and has even symmetry with respect to m . By symmetry, this linear multiwavelet is orthogonal to the slopes; the height is chosen so that it is orthogonal to the constants, too. We call this a *vee*.
- On each dyadic interval I as above, find a vector that is linear on each of L and R and is orthogonal to the constants, slopes, and the vee on I . This one is called an *odd*. (It has odd symmetry about the midpoint of I . It is the only vector of the four that is disconnected in the interior of its support.)

We now confirm that the above algorithms can be modified in a straightforward way to produce piecewise-linear representations. We need to check each of the following easy facts.

- Each data index is in $O(\log(N))$ dyadic intervals of length at least 4, so is in the support of $O(\log(N))$ linear multiwavelets.
- Each multiwavelet vector is a piecewise-linear vector of $O(1)$ pieces.
- Each element of a piecewise linear representation (i.e., for each bucket j , the vector that results from setting to zero all heights and slopes *other* than the j th height and slope) is the linear combination of the $O(\log(N))$ linear multiwavelets whose support intersects an endpoint of the bucket.
- The transform that takes N data points to the N linear multiwavelet coefficients is expressible as a tree of basic filters, like in the case of Haar wavelets. For linear multiwavelets, the basic filters have four inputs and four outputs and act according to the matrix in Fig. 5. The vee and odd are sent to the overall output and the constant and slope become input to another basic filter. At the leaf-level of the tree, we need special 2-input, 2-output filters that take individual data items as inputs and compute a constant and a slope as input to the 4-input, 4-output basic filters.
- We can build the best B -bucket piecewise-linear representation \mathbf{H} to a sparse linear multiwavelet representation \mathbf{R}_{rob} using the technique of Sect. 3.2; the time and space are comparable.

Finally, we note that these techniques can be used more directly to produce linear multiwavelet representations. While these are less common than Haar wavelets, histograms, or piecewise-linear representations, they are a natural variant in our context.

6.2 Range Queries

To this point, we have discussed only building histograms that are nearly optimal for *point* queries. We now consider *range* queries. That is, at the user level, a query is a pair of points (ℓ, r) and the ideal answer is $\mathbf{A}[\ell, r] = \sum_{\ell \leq i < r} \mathbf{A}[i]$. An answer from a (piecewise-constant) histogram \mathbf{H} would be $\mathbf{H}[\ell, r] = \sum_{\ell \leq i < r} \mathbf{H}[i]$. The sum square error of a histogram is

$$\sum_{\ell, r} |\mathbf{A}[\ell, r] - \mathbf{H}[\ell, r]|^2.$$

Note that, for any \mathbf{H} , the empty query $[\ell, \ell]$ is always answered perfectly as 0. We take the convention that $\mathbf{A}[r, \ell] = -\mathbf{A}[\ell, r]$. It then follows that we may equivalently take the sum over all queries (ℓ, r) or require that $\ell < r$ or $\ell \leq r$; the sum square error corresponds exactly.

Building histograms for range queries is less straightforward than building histograms for point queries. For point or range queries, note that the data can essentially fix the bucketing. For example, if the data alternates between runs of values between 0 and 10 and runs of values between 1,000,000 and 1,000,010, then

the bucketing must respect the runs of data—anything else would be awful. For range queries, even for a fixed bucketing, it is not obvious how to choose the bucket heights. This is because the optimal height for bucket j depends on other bucket heights and data outside the bucket—for example, if buckets $j - 1$ and $j + 1$ underestimate the data, we may want bucket j to overestimate the data, to give net error contribution zero of the three buckets to overflying queries. Within a single bucket, there is a tension between making the bucket height equal to the average of the data in the bucket (to avoid contributing error to overflying queries) and setting the height to optimize within-bucket queries. For example, if the data in a bucket of size s is all zero except for a -1 near an endpoint and a $+1$ near the center, then the average data value in the bucket is 0. But roughly half the intra-bucket range queries have answer $+1$, roughly half have answer 0, and $O(s)$ of the $\Theta(s^2)$ intra-bucket queries have answer -1 , so, from the perspective of intra-bucket queries, the ideal height will be positive. Unlike the situation for point queries, we know of no polynomial time algorithm, even using full space, that finds the optimal B -bucket histogram for range queries.

Nevertheless, we can easily transform this problem so that the previously described approximation algorithms will work on it. We will need the notion of a *connected* piecewise-linear representation. The definition is standard for continuous-domain functions; we will say that an integer-domain function is B -piece piecewise-linear and connected if it is the integer sampling of a B -piece piecewise-linear continuous-domain function. For example, $\langle 2, 4, 6, 8, 7, 6, 5 \rangle$ is a 2-piece piecewise-linear connected representation whereas $\langle 1, 2, 3, 8, 7, 6, 5 \rangle$ is 2-piece piecewise-linear but not connected.

To exploit the above algorithms, we take the original data, \mathbf{A} , and transform to the prefix array, $\mathcal{P}(\mathbf{A})$. Under this transform, a piecewise-constant representation transforms to a piecewise linear connected representation and the square error summed over all range queries (ℓ, r) transforms to N times the sum square error over all point queries. That is, to get a piecewise-constant representation that minimizes the sum square error of range queries to \mathbf{A} , we only need to find a piecewise linear connected representation \mathbf{H} that minimizes the sum square error of point queries to $\mathcal{P}(\mathbf{A})$, then output $\Delta(\mathbf{H})$, where $\Delta(\mathbf{H})[i] = \mathbf{H}[i + 1] - \mathbf{H}[i]$. (Thus Δ and \mathcal{P} are opposite operations.) Also note that, if the length of \mathbf{A} is N , then the length of $\mathcal{P}(\mathbf{A})$ is $N + 1$ and $\mathcal{P}(\mathbf{A})[0]$ is always 0.

Unfortunately, we do not know how to find optimal piecewise linear *connected* representations—the transformed problem is completely equivalent to the original. But we do know how to find nearly optimal representations from the larger class general piecewise-linear B -bucket representations, not necessarily connected. So the algorithm is

- Convert \mathbf{A} to $\mathcal{P}(\mathbf{A})$, on the fly.
- Find a (near-) best piecewise linear representation \mathbf{H} for $\mathcal{P}(\mathbf{A})$.
- Output $\Delta(\mathbf{H})$.

Formally, recall that, if $|\mathbf{A}| = N$, then $|\mathcal{P}(\mathbf{A})| = N + 1$ and $\mathcal{P}(\mathbf{A})[0] = 0$; we find a piecewise-linear representation only for the N values $\mathcal{P}(\mathbf{A})[1, N]$. Furthermore, one

can show that if the bucket heights are optimal in the piecewise-linear representation (it is easy to make both the heights and slopes optimal), then the correspondence is preserved between the sum square error over point queries to piecewise-linear \mathbf{H} and the sum square error over range queries to $\Delta(\mathbf{H})$. Finally, we consider $\Delta(\mathbf{H})$. Since \mathbf{H} is not connected, $\Delta(\mathbf{H})$ is not necessarily a B -bucket piecewise-constant representation, but it is a $(2B - 1)$ -bucket piecewise-constant representation, where the discontinuities in \mathbf{H} each give rise to a bucket of size one in $\Delta(\mathbf{H})$. For example, if \mathbf{H} is given by $\langle 1, 2, 3, 8, 7, 6, 5 \rangle$, then $\Delta(\mathbf{H})$ is $\langle 1, 1, 5, -1, -1, -1 \rangle$.

Next, we consider the changes necessary to our algorithms in order to realize the desired transform. It is easy to perform the Δ operation in post-processing on a small object like \mathbf{H} . As for pre-processing, in the ordered aggregate model, we can easily convert \mathbf{A} to $\mathcal{P}(\mathbf{A})$ in time $O(1)$ per item by a pipe before proceeding to compute linear multiwavelet coefficients. In the dynamic model, note that an update to $\mathbf{A}[i]$ transforms to an update to each $\mathcal{P}(\mathbf{A})[j]$ for $j > i$, i.e., to the function $\chi_{[i+1, N+1]}$. In turn, this causes an update to several linear multiwavelets, but only those whose support intersects an endpoint of $[i + 1, N + 1]$, i.e., the $O(\log(N))$ of the linear multiwavelets whose support contains $i + 1$. The rest of the algorithms are unchanged.

It follows that we can get a

1. $(2, 1)$ -approximation algorithm in polynomial time and full space using the techniques of Sect. 3.1;
2. $(2, 1 + \epsilon)$ -approximation algorithm for ordered aggregate data in space $(B \log(N) \log(M)/\epsilon)^{c_2}$ and time $c_1 N + (B \log(N) \log(M)/\epsilon)^{c_2}$ using the techniques of Sect. 4.3; and a
3. $(2, 1 + \epsilon)$ -approximation algorithm for dynamic data in time and space $(B \log(N) \log(M)/\epsilon)^{c_2}$ using the techniques of Sect. 5.

Finally, note that, because the resulting $2B$ -bucket histogram has alternate buckets of size 1, we only need to store B boundaries and $2B$ heights. Arguably, the storage requirement is only 1.5 times that of a general B -bucket histogram.

We now describe briefly a $(1, 1 + \epsilon)$ -approximation in polynomial time and full space using an offline algorithm. First we build a robust representation \mathbf{R}_{rob} for $\mathcal{P}(\mathbf{A})$ in linear time, as above. Then we want to find the best piecewise-constant histogram for range queries to $\Delta(\mathbf{R}_{\text{rob}})$. We now show how to do this, modifying the techniques of Sect. 3.2.

The dynamic program of Sect. 3.2 asked, “What is the greatest prefix $[0, m]$ that has a representation with ℓ buckets and error budget approximately k (in units of δ)?” Knowing the answer for a given ℓ and all k lets us answer the question for $\ell + 1$ and all k . The computational cost depends on the range and precision needed for the error budget; in Sect. 3.2, the error could be at most $2E_{\text{opt}}$ and units of $\delta \geq \Omega(\epsilon E_{\text{opt}}/B)$ suffice.

To generalize for range queries, we will need to track an additional quantity. Specifically, define the sum suffix error of a partial histogram \mathbf{H} on $[0, m)$ to be $\sum_{i < m} (\mathbf{A}[i, m) - \mathbf{H}[i, m))$. Observe that if we know the sum suffix error σ and the sum square error E for a prefix $[0, m)$, that, together with local information in

a potential new bucket $[m, m')$, suffices to compute the two quantities for the new prefix $[0, m')$. Specifically, the new sum suffix error is a linear combination of σ , the overflight error $A[m, m') - H[m, m')$, and the sum suffix error σ'_1 within the bucket $[m, m')$; the new sum square error is a linear combination of E , the sum square error of queries within $[m, m')$, and $\sigma_1\pi_1$, where π_1 is the analogously-defined sum prefix error within $[m, m')$. To see this, one can trace the contribution to the three quantities of a single range query, whose endpoints may be in $[0, m)$, $[m, m')$, or $[m', N)$. The coefficients in the linear combinations mentioned above depend only on m, m' , and N ; they account for the number of ranges making contributions.

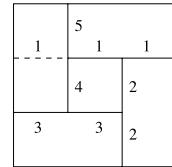
So, by analogy with Sect. 3.2, we ask, “What is a prefix $[0, m)$ that has a representation with ℓ buckets, sum suffix error budget approximately i , and sum square error budget approximately j ?” As above, j needs to take values up to $O(E_{\text{opt}})$ in units of $\mathcal{Q}((\epsilon/B)E_{\text{opt}})$. By the Cauchy–Schwartz inequality, one can show that the sum suffix error is at most $\sqrt{2NE_{\text{opt}}}$, since, otherwise, these at-most- N suffixes would contribute more than $2E_{\text{opt}}$ to the total sum square error. Observe that the sum suffix error ultimately contributes to the sum square error only by multiplying a sum prefix error, which is also bounded by $\sqrt{2NE_{\text{opt}}}$. It follows that we only need precision $\mathcal{Q}((\epsilon/B)\sqrt{E_{\text{opt}}/N})$ for sum suffix errors in order that the B quantization errors total $O(B \cdot ((\epsilon/B)\sqrt{E_{\text{opt}}/N}) \cdot \sqrt{NE_{\text{opt}}}) = O(\epsilon E_{\text{opt}})$. Thus there are $O(\sqrt{NE_{\text{opt}}} / ((\epsilon/B)\sqrt{E_{\text{opt}}/N})) = O(BN/\epsilon)$ possible values for i , a number polynomial in BN/ϵ , as desired. Since there are B/ϵ possibilities for m and B possibilities for ℓ , the total space is polynomial in B, N , and ϵ . Similarly, the time is polynomial in B, N, ϵ , and $\log \log M$, where M is bound on the range of values. Improvements to the running time will be of great interest. These results for rangesum appear in [11].

6.3 Multidimensional Histograms

Now we generalize the basic problem to multi-, say two-, dimensional arrays. The first issue that arises is what is meant by a two-dimensional histogram summary. One can think of partitions into rectangles, squares and other shapes, or covers (i.e., allow cells to overlap) in which case we have a variety of ways to interpret the approximation in overlapped areas. Histograms with overlapping cells are discussed in [14] where the authors present polylogarithmic space algorithms, albeit with $\mathcal{Q}(N^2)$ time. Here, we focus on partition-based histograms. Among these, there are a number of variations depending upon how the partition is done. The natural and general class is one of arbitrary partitioning into B axis-parallel rectangles.

Such two (and higher) dimensional problems are known to be NP-hard, but, in two dimensions, a $(O(1), 1)$ -approximation can be computed in polynomial time and space [10]. For example, the authors in [10] give an algorithm to produce the best *hierarchical* histogram. A *hierarchical* histogram is one with a hierarchical bucketing, i.e., in which the bucketing is obtained by repeatedly partitioning one of the existing buckets into two pieces. Since any bucketing of B buckets in two

Fig. 6 A 5-bucket non-hierarchical partition refined (by dashed line) into a 6-bucket partition histogram. The five hierarchical cuts may be made in the order indicated



dimensions can be refined to a hierarchical bucketing of $4B$ buckets [1], a best $4B$ -bucket hierarchical histogram is at least as good as the best B -bucket general histogram, so we get a $(4, 1)$ -approximation to the general problem; see Fig. 6.

In streaming context, consider $N \times N$ integer-valued signal \mathbf{A} . One can solve the B -bucket, $(4, 1 + \epsilon)$ -approximate two dimensional histogram problem using $(B \log \|\mathbf{A}\| \log(N)/\epsilon)^{O(1)}$ space, per-item time, and time per hist query. For static data, for constants c_1 and c_2 , we can $(4, 1 + \epsilon)$ -approximate the best B -bucket histogram in time $c_1 N^2 + (B \log \|\mathbf{A}\| \log(N)/\epsilon)^{c_2}$ and space $(B \log \|\mathbf{A}\| \log(N)/\epsilon)^{c_2}$, making one pass over the data, with the order we specify to lay out \mathbf{A} . Both the results above are achieved by using techniques that have been developed in the context one dimensional histograms [6, 7], combining them with techniques known for offline multidimensional histogramming [10], and extending them. The technical details are in [12].

References

1. F. d'Amore, P.G. Franciosa, On the optimal binary plane partition for sets of isothetic rectangles. *Inf. Process. Lett.* **44**(5), 255–259 (1992)
2. M. Blum, R. Floyd, V. Pratt, R. Rivest, R. Tarjan, Time bounds for selection. *J. Comput. Syst. Sci.* **7**, 448–461 (1972)
3. G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, in *LATIN* (2004), pp. 29–38
4. H. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, T. Suel, Optimal histograms with quality guarantees, in *Proc. of the 1998 Intl. Conf. on Very Large Data Bases (VLDB)* (1998), pp. 275–286
5. A. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *Proc. of the 2001 Intl. Conf. on Very Large Data Bases (VLDB)* (2001), pp. 79–88
6. A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M. Strauss, Fast, small-space algorithms for approximate histogram maintenance, in *Proc. STOC* (2002), pp. 389–398
7. S. Guha, P. Indyk, S. Muthukrishnan, M. Strauss, Histogramming data streams with fast per-item processing, in *Proc. ICALP* (2002), pp. 681–692
8. S. Guha, N. Koudas, K. Shim, Data-streams and histograms, in *Proc. of the 2001 Annual ACM Symp. on Theory of Computing (STOC)* (2001), pp. 471–475
9. S. Guha, N. Koudas, Approximating a data stream for querying and estimation: algorithms and performance evaluation, in *Proc. of the 2002 Intl. Conf. on Data Engineering (ICDE)* (2002), pp. 567–576
10. S. Muthukrishnan, V. Poosala, T. Suel, On rectangular partitionings in two dimensions: algorithms, complexity, and applications, in *Proc. ICDT* (1999), pp. 236–256
11. S. Muthukrishnan, M. Strauss, in *Rangesum Histograms*. Proc. ACM-SIAM SODA (2003), pp. 233–242

12. S. Muthukrishnan, M. Strauss, Maintenance of multidimensional histograms, in *Proc. FSTTCS* (2003), pp. 352–362
13. G. Strang, V. Strela, Orthogonal multiwavelets with vanishing moments, in *Proc. SPIE*, ed. by H.H. Szu. Wavelet Applications, vol. 2242 (1994), pp. 2–9
14. N. Thaper, S. Guha, P. Indyk, N. Koudas, Dynamic multidimensional histograms, in *Proc. ACM SIGMOD Conference* (2002), pp. 428–439

Stable Distributions in Streaming Computations

Graham Cormode and Piotr Indyk

1 Introduction

In many streaming scenarios, we need to measure and quantify the data that is seen. For example, we may want to measure the number of distinct IP addresses seen over the course of a day, compute the difference between incoming and outgoing transactions in a database system or measure the overall activity in a sensor network. More generally, we may want to cluster readings taken over periods of time or in different places to find patterns, or find the most similar signal from those previously observed to a new observation. For these measurements and comparisons to be meaningful, they must be well-defined. Here, we will use the well-known and widely used L_p norms. These encompass the familiar Euclidean (root of sum of squares) and Manhattan (sum of absolute values) norms.

In the examples mentioned above—IP traffic, database relations and so on—the data can be modeled as a vector. For example, a vector representing IP traffic grouped by destination address can be thought of as a vector of length 2^{32} , where the i th entry in the vector corresponds to the amount of traffic to address i . For traffic between (source, destination) pairs, then a vector of length 2^{64} is defined. The number of distinct addresses seen in a stream corresponds to the number of non-zero entries in a vector of counts; the difference in traffic between two time-periods, grouped by address, corresponds to an appropriate computation on the vector formed by subtracting two vectors, and so on. As is usual in streaming, we assume that the domain

G. Cormode (✉)

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

e-mail: G.Cormode@warwick.ac.uk

P. Indyk

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA

e-mail: indyk@mit.edu

of the data and the size of the data are too massive to permit the direct computation of the functions of interest—which are otherwise mostly straightforward—and instead, we must use an amount of storage that is much smaller than the size of the data. For the remainder of this chapter, we put our description in terms of vectors, with the understanding that this is an abstraction of problems coming from a wide variety of sources.

Throughout, we shall use \mathbf{a} and \mathbf{b} to denote vectors. The dimension of a vector (number of entries) is denoted as $|\mathbf{a}|$.

Definition 1 The L_p norm (for $0 < p \leq 2$) of a vector \mathbf{a} of dimension n is

$$\|\mathbf{a}\|_p = \left(\sum_{i=1}^n |\mathbf{a}[i]|^p \right)^{1/p}.$$

The L_0 norm is defined as

$$\|\mathbf{a}\|_0 = \left(\sum_{i=1}^n |\mathbf{a}[i]|^0 \right) = |\{i \mid \mathbf{a}[i] \neq 0\}|$$

where 0^0 is taken to be 0.

These are vector norms in the standard sense: the result is non-negative, and zero only when the vector is zero; and the norm of the sum of vectors is less than the sum of their norms. For $p > 0$, the L_p norm guarantees that $\|k\mathbf{a}\|_p = k\|\mathbf{a}\|_p$ for any scalar k . This does not hold for $p = 0$, so L_0 is not a norm in the strict sense. These norms immediately allow the measurement of the difference between vectors, by finding the norm of the (component-wise) difference between them. To be precise,

Definition 2 The L_p distance between vectors \mathbf{a} and \mathbf{b} of dimension n is the L_p norm of their difference,

$$\|\mathbf{a} - \mathbf{b}\|_p = \left(\sum_{i=1}^n |\mathbf{a}[i] - \mathbf{b}[i]|^p \right)^{1/p}.$$

The L_p distance encompasses three very commonly used distance measures:

- Euclidean distance, given by L_2 distance, is the root of the sum of the squares of the differences of corresponding entries.
- The Manhattan distance, given by the L_1 distance, is the sum of the absolute differences.
- The Hamming distance, given by the L_0 distance, is the number of non-zero differences.

In this chapter, we will show how all three of the distances can be estimated for massive vectors presented in the streaming model. This is achieved by making

succinct *sketches* of the data, which can be used as synopses of the vectors they summarize. This is described in Sect. 2. In Sect. 3, we discuss some applications of these results, to the distinct elements problem, and to computing with objects that can't be modeled as simple vectors. Lastly, we discuss related work and new directions in Sects. 4 and 5.

2 Building Sketches Using Stable Distributions

2.1 Data Stream Model

We assume a very general, abstracted model of data streams where our input arrives as a stream of updates to process. We consider vectors $\mathbf{a}, \mathbf{b}, \dots$, which are presented in an implicit, incremental fashion. Each vector has dimension n , and its current state at time t is $\mathbf{a}(t) = [\mathbf{a}(t)[1], \mathbf{a}(t)[2], \dots, \mathbf{a}(t)[n]]$. For convenience, we shall usually drop t and refer only to the current state of the vector. Initially, \mathbf{a} is the zero vector, $\mathbf{0}$, so $\mathbf{a}(0)[i]$ is 0 for all i . Updates to individual entries of the vector are presented as a stream of pairs. The t th update is (i_t, c_t) , meaning that

$$\begin{aligned}\mathbf{a}(t)[i_t] &= \mathbf{a}(t-1)[i_t] + c_t, \\ \mathbf{a}(t)[j] &= \mathbf{a}(t-1)[j], \quad j \neq i_t.\end{aligned}$$

For the most part, we expect the data to arrive in no particular order, since it is unrealistic to expect it to be sorted on any attribute. We also assume that each index can appear many times over in the stream of updates. In some cases, c_t s will be strictly positive, meaning that entries only increase; in other cases, c_t s are allowed to be negative also. The former is known as the *cash register* case and the latter the *turnstile* case [35]. Here, we assume the more general case, that the data arrives unordered and each index can be updated multiple times within the stream.

2.2 Stable Distributions

Stable Distributions are a class of statistical distributions with properties that allow them to be used in finding L_p norms. This allows us to solve many problems of interest on data streams. A stable distribution is characterized by four parameters (following [38]), as follows:

- The stability parameter $\alpha \in (0, 2]$,
- The skewness parameter $\beta \in [-1, 1]$,
- The scale parameter $\gamma > 0$,
- The shift parameter δ .

Although there are different parameterizations of stable distributions, we shall fix values of β , γ and δ for this discussion. This has the effect that the different parameterization systems all coincide. We set $\beta = 0$, which makes the distribution symmetric about its mode. Setting $\gamma = 1$ and $\delta = 0$ puts the mode of the distribution at 0 and gives a canonical distribution. Formally then, the distributions we consider are *symmetric and strictly stable*, but we shall simply refer to them as stable.

Definition 3 A (*strictly*) *stable distribution* is a statistical distribution with parameter α in the range $(0, 2]$. For any three independent random variables X, Y, Z drawn from such a distribution, for scalars a, b , $aX + bY$ is distributed as $(|a|^\alpha + |b|^\alpha)^{1/\alpha} Z$.

These are called stable distributions because the underlying distribution remains stable as instances are summed: the sum of stable distributions (with the same α) remains stable. This can be thought of a generalization of the central limit theorem, which states that the sum of distributions (with finite variance) will tend to a Gaussian distribution. Note that, apart from $\alpha = 2$, stable distributions have unbounded variance.

Several well-known distributions are known to be stable. The Gaussian (normal) distribution, with density $f(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$, is strictly stable with $\alpha = 2$. The Cauchy distribution, with density $f(x) = \frac{1}{\pi(1+x^2)}$, is strictly stable with $\alpha = 1$. For all values of $\alpha \leq 2$, stable distributions can be simulated by using appropriate transformations from uniform distributions, as we will show later.

2.3 Sketch Construction

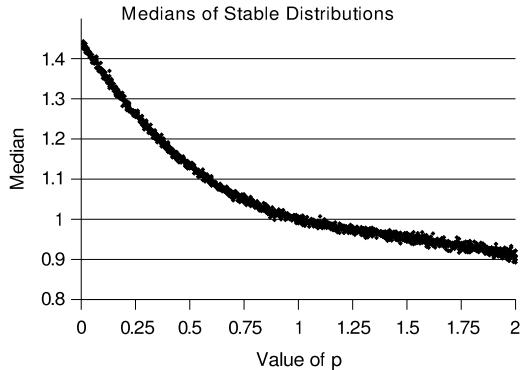
By applying the above definition iteratively, we find that

Corollary 1 Given random variables X_1, X_2, \dots, X_n independently and identically distributed as X , a strictly stable distribution with stability parameter $\alpha = p$, and a vector \mathbf{a} , then $S = \mathbf{a}[1]X_1 + \mathbf{a}[2]X_2 + \dots + \mathbf{a}[n]X_n$ is distributed as $\|\mathbf{a}\|_p X$.

From this corollary, we get the intuition for why stable distributions are helpful in computing L_p norms and L_p distances: by maintaining the inner product of variables from α -stable distributions with a vector being presented in the stream, we get a variable S which is distributed as a stable distribution scaled by the L_p norm of the stream, where $p = \alpha$. Maintaining this inner product as the vector undergoes updates is straightforward: given an update (i, c) , we simply add $X_i \cdot c$ to S . However, what we have so far is an equality *in distribution*; what we are aiming for is an equality *in value*. To solve this problem, we proceed as follows.

Let $\text{med}(X)$ denote the *median* of X , i.e., a value M such that $\Pr[X > M] = 1/2$. Then, for any $s > 0$, we have $\text{med}(s \cdot |X|) = s \cdot \text{med}(|X|)$. In our case, $\text{med}(|S|) =$

Fig. 1 Plot of empirically found median of stable distributions varying stability parameter p in the range 0 to 2



$\text{med}(\|\mathbf{a}\|_p \cdot |X|) = \|\mathbf{a}\|_p \text{med}(|X|)$. Moreover, $\text{med}(|X|)$ depends only on p and can be precomputed in advance. For $\alpha = 1$ and $\alpha = 2$ then $\text{med}(|X|) = 1$. For other values of α , $\text{med}(|X|)$ can be found numerically: Fig. 1 shows a plot of the median values found by simulation, where each point represents one experiment of taking the median of 10,000 drawings, raised to the power $\frac{1}{p}$. Thus, in order to estimate $\|\mathbf{a}\|_p$, it suffices to compute an estimate z of $\text{med}(|S|)$, and then estimate $\|\mathbf{a}\|_p$ by $\frac{z}{\text{med}(|X|)}$.

To estimate $M = \text{med}(|S|)$, we take a vector $sk[1] \dots sk[m]$ of independent samples of the random variable S . In other words, for each i , we “generate” m independent samples x_i^1, \dots, x_i^m of the X_i , and then compute $sk[j] = \mathbf{a}[1]x_1^j + \dots + \mathbf{a}[n]x_n^j$. We call this vector sk an α -stable sketch of the vector \mathbf{a} . This can be viewed computationally as maintaining each entry of the sketch as the inner product between the vector and appropriately chosen random vectors, i.e., $sk[j] = \mathbf{a} \cdot x^j$. The procedure is presented in more detail in Fig. 2. Note that the vector sk can be computed in a streaming fashion; in particular, the numbers x_i^j are not actually stored. It is important that we get the same value for x_i^j every time it is accessed: this is done by using a pseudo-random number generator that is initialized with i to give a stream of values x_i^j . Then we use the following lemma (for the random variable $Z = |S|$, the absolute value of S).

Lemma 1 *For any one-dimensional random variable Z with continuous density, let $F(t) = \Pr[Z \leq t]$. There is a constant $C > 0$ such that for $m = \frac{C}{\epsilon^2} \log \frac{2}{\delta}$, if we take m independent samples z_1, \dots, z_m of Z and set z to be the median element of the sequence z_1, \dots, z_m , then*

$$\Pr\left[F(z) \in \left[\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon\right]\right] > 1 - \delta.$$

Proof Let t be such that $F(t) = \frac{1}{2} - \epsilon$. If $F(z) < F(t) = \frac{1}{2} - \epsilon$, then $z_i < t$ for most z_i 's, and therefore $\frac{|\{i: z_i \geq t\}|}{m} < \frac{1}{2}$. However, for each z_i we have $\Pr[z_i \geq t] = \frac{1}{2} + \epsilon$. Therefore, from Chernoff bound [34] we know that there exists a constant

$C > 0$ such that

$$P_1 = \Pr\left[F(z) < \frac{1}{2} - \epsilon\right] \leq \exp\left(-\frac{\epsilon^2 m}{C}\right).$$

Using the same argument, we obtain

$$P_2 = \Pr\left[F(z) > \frac{1}{2} + \epsilon\right] \leq \exp\left(-\frac{\epsilon^2 m}{C}\right).$$

By setting $m = \frac{C}{\epsilon^2} \log(2/\delta)$, we obtain $P_1 + P_2 \leq \delta$. \square

Note that our goal is to obtain an approximation to the median M , i.e., the number such that $F(M) = \frac{1}{2}$, while the above provides us (with probability $1 - \delta$) with z such that $F(z) \in [\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon]$. For general functions F , z could be a bad estimate of M ; e.g., if F is “flat” around the point $\frac{1}{2}$. However, if the derivative F' of F is bounded by $\frac{1}{B}$ from below around $\frac{1}{2}$, then the above implies that $z \in [M - B\epsilon, M + B\epsilon]$, i.e., that $z = (1 \pm B\epsilon)M$, which is precisely what we want.

It suffices to verify if the derivative of the function F for the random variable $|S|$ is bounded away from 0 around $\frac{1}{2}$. For $\alpha = \{1, 2\}$, this can be verified analytically. For other values of α , this can be verified computationally, e.g., by plotting F . It should be noted that the lower bound for $F'(\frac{1}{2})$ depends on α , and tends to ∞ as α tends to 0.

2.4 Simulating Stable Distributions

When implementing this technique, we need to be able to generate values from a stable distribution. These can be generated by using appropriate transformations from uniform random distributions.

- For $\alpha = 1$, we can use the Cauchy distribution, which is easy to draw from. If U is a uniform random distribution returning values in the range $[0, 1]$, then $\tan(\pi(U - \frac{1}{2}))$ is distributed with the Cauchy distribution.
- For $\alpha = 2$, we can use the Normal distribution, which can be drawn from using the Box–Muller transformation: If U and V are independently distributed uniformly over $[0, 1]$, then $\sqrt{-2 \ln U} \cos(2\pi V)$ is distributed as a normal distribution.
- For all other values of $\alpha \in (0, 2)$, stable distributions can be simulated using the method of Chambers, Mallows and Stuck [6]. These take uniform distributions U, V onto the range $[0, 1]$ and output a value drawn from a stable distribution with parameter $\alpha \neq 1$. Set $\theta(U) = \pi \cdot (U - \frac{1}{2})$. Then

$$\text{stable}(U, V, \alpha) = \frac{\sin \alpha \theta(U)}{\cos^{1/\alpha} \theta(U)} \left(\frac{\cos(\theta(U) \cdot (1 - \alpha))}{-\ln V} \right)^{\frac{1-\alpha}{\alpha}}$$

is distributed as a stable distribution with parameter α .

Algorithm to compute sketches of a stream

```

1: for  $1 \leq j \leq m$  do
2:    $sk[j] \leftarrow 0.0$ 
3: for all tuples  $(i, c)$  do
4:   initialize-random-seed( $i$ )
5:   for  $1 \leq j \leq m$  do
6:      $u \leftarrow$  uniformly-random-from( $0, 1$ )
7:      $v \leftarrow$  uniformly-random-from( $0, 1$ )
8:      $x_i^j \leftarrow$  stable( $u, v, \alpha$ )
9:      $sk[j] \leftarrow sk[j] + c \cdot x_i^j$ 
10: return median( $|sk[1]|, \dots, |sk[m]|$ ) / med( $|X|$ )

```

Fig. 2 Sketching algorithm

2.5 The Sketch Algorithm

The full algorithm to compute a sketch of a stream is given in Fig. 2. It works as follows: lines 1–2 initialize the sketch vector to a vector of all zeros. Then for each new tuple (i, c) , we initialize a pseudo-random number generator with the index i (line 4), so that when we draw random values (lines 6–7), these are pseudo-random functions of i , the same every time the same value of i is seen in the stream. Each successive call to the random number generator yields a new value, but the sequence of values following each re-initialization is the same. Line 8 takes two values in the range 0 to 1, and transforms them to yield a value drawn from a stable distribution with parameter α . The j th entry of the sketch is updated, by adding on the contribution of the update c times the stable value in line 9. This is repeated for all m entries in the sketch. Lastly, to return an estimate of the norm of the vector, we take the median of the (absolute) values of the sketch, and scale this by the median of the stable distribution with parameter α .

We state a theorem that summarizes the properties of this algorithm.

Theorem 1 ([26]) *In space $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ we can compute an α -stable sketch of a vector \mathbf{a} presented in the turnstile streaming model. Using this sketch we can compute an estimate of $\|\mathbf{a}\|_p$ for $p = \alpha$ that is accurate within a factor of $1 \pm \epsilon$ with probability at least $1 - \delta$. Processing each update to the vector \mathbf{a} takes time linear in the size of the sketch, $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$.*

This follows from the above lemma and the preceding discussion.

Note that to complete the proof we must also argue that we can replace truly random samples x_i^j with values drawn using pseudo-random generators. The proof of this relies on the pseudo-random generators of Nisan [37], and we refer the interested reader to the details in [26]. In practice, it suffices to use standard random number generators to generate uniform pseudo-random numbers, and use the transforms given in the previous section.

2.6 Other Estimators

In the previous sections we used the median of $sk[1], \dots, sk[m]$ to estimate the norm of the stream vector. There are alternative estimators that one can use instead. In particular, for the L_1 norm, Li et al. [32] proposed the following *bias-corrected geometric mean* estimator:

$$E = \cos^m \left(\frac{\pi}{2m} \right) \prod_{j=1}^m |sk[j]|^{1/m}.$$

This estimator is more accurate than the median estimator when the sample size is small [32].

A similar estimator can be used to estimate the more general L_p norms, $p \in (0, 2]$. Unlike the median estimator (which requires some computation to determine the right parameters of the distribution function F), the geometric mean estimator is computable using a simple analytical formula; see [31] for more details.

2.7 Combining Sketches

We now state a number of the properties of this sketching technique, which follow immediately from the method of their construction. These show how the α -stable sketches have application to a variety of circumstances.

Corollary 2

$$\begin{aligned} sk(\mathbf{a} + \mathbf{b}) &= sk(\mathbf{a}) + sk(\mathbf{b}), \\ sk(\mathbf{a} - \mathbf{b}) &= sk(\mathbf{a}) - sk(\mathbf{b}). \end{aligned}$$

These two facts follow immediately from the fact that the sketches are generated as the inner product between the vector \mathbf{a} and vectors of values drawn from random distributions, x_i^j . So, the sketch of the sum of two vectors can be computed from the sum of their sketches. This allows the distributed computation of sketches by multiple parties: after agreeing in advance on a random number generator to use, sketches of different data can be computed in parallel, and then the sketches combined to get the sketch of the sum of the data. Similarly, the sketch of the difference of two vectors, and hence the L_p distance between them, can be computed from sketches of the original vectors. This allows large data sets to be compared by only storing the short summarizing sketches of them.

Corollary 3

$$sk(c \cdot \mathbf{a}) = c \cdot sk(\mathbf{a}).$$

Also by the linearity of construction, the sketch of a vector \mathbf{a} scaled by a scalar c can be computed directly from the sketch of the original vector. This allows, for example, a new day's set of data to be compared against the *average* of the previous weeks data: the sketch of the average is computed by summing the sketches of seven days data, and scaling by $\frac{1}{7}$. Similarly, the popular *exponential decay* model where we compute a weighted average of previous vectors $\mathbf{a}(0), \mathbf{a}(1), \mathbf{a}(2), \dots$ as $(1 - \lambda)(\mathbf{a}(0) + \lambda\mathbf{a}(1) + \lambda^2\mathbf{a}(2) + \dots + \lambda^i\mathbf{a}(i) + \dots)$ ($0 < \lambda < 1$) is easy to construct iteratively. Suppose we have a sketch of the current vector sk , and wish to include \mathbf{a} as the new day's data. Then we can set $sk[j] \leftarrow (1 - \lambda)sk(\mathbf{a})[j] + \lambda sk[j]$ for all j .

3 Application to Streaming Problems

In this section, we outline some of the applications within streaming and beyond that stable distributions have been used to address. These include: estimating the number of distinct items in a stream; as a way to track embeddings in small space; and for geometric problems such as clustering and approximate nearest neighbor searching.

3.1 L_0 and Counting Distinct Items

Suppose we are shown a sequence of items, and want to know how many *distinct* items there are in the sequence. This is a fundamental question in data stream analysis, and it has a large number of applications both in this form and for generalizations of this problem. Assume that each item is an integer in the range $1..n$. Then we could maintain a vector \mathbf{a} where $\mathbf{a}[i]$ counts the number of occurrences of item i . Arrivals of new items can be modeled as adding one to the appropriate entry in the vector. In the turnstile streaming model, departures can be modeled as subtracting one from the corresponding entry. The number of distinct items corresponds to the L_0 norm \mathbf{a} , that is, the number of non-zero counts. The L_0 norm is somewhat more general than this, since it can also incorporate negative counts. Such negative counts arise, for example, when we want to compare two vectors of counts, and find in how many places the counts differ (the Hamming difference). Note that stable distributions do not exist for $\alpha = 0$, so we cannot directly apply the sketching technique. Instead, we observe that for sufficiently small values of p , the L_p norm approximates the L_0 norm:

Theorem 2 ([11]) *The L_0 norm $\|\mathbf{a}\|_0$ can be approximated by finding the L_p norm of the integer valued vector \mathbf{a} for sufficiently small p ($0 < p \leq \frac{\epsilon}{\log U}$) provided we have an upper bound (U) on the size of each entry in the vector, so $\forall i : |\mathbf{a}[i]| < U$.*

Proof We show that the L_0 norm of a vector can be well-approximated by $\sum_i |\mathbf{a}[i]|^p = \|\mathbf{a}\|_p^p$ for a small value of p ($p > 0$). If, for all i we have that $|\mathbf{a}[i]| \leq U$

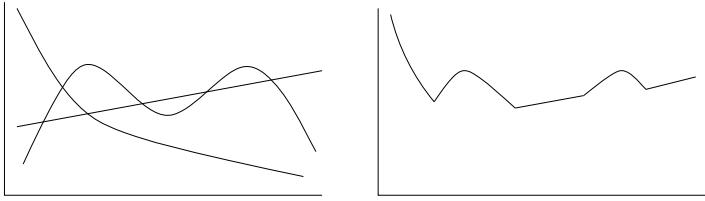


Fig. 3 The “dominance norm” of multiple signals (*left*) computes the “area under the curve” of the upper envelope of multiple signals (*right*)

for some upper bound U , then

$$\begin{aligned}\|\mathbf{a}\|_0 &= \sum_i |\mathbf{a}[i]|^0 \leq \sum_i |\mathbf{a}[i]|^p \leq \sum_i U^p |\mathbf{a}[i]|^0 \\ &\leq U^p \sum_i |\mathbf{a}[i]|^0 \leq (1 + \epsilon) \sum_i |\mathbf{a}[i]|^0 = (1 + \epsilon) \|\mathbf{a}\|_0.\end{aligned}$$

We use the fact that $\mathbf{a}[i]$ is an integer and $\forall i : |\mathbf{a}[i]| \leq U$. The last inequality uses $U^p \leq (1 + \epsilon)$ which follows if we set $p \leq \frac{\ln(1+\epsilon)}{\ln U} \approx \frac{\epsilon}{\ln U}$. \square

From this, it follows that if we set the α of our sketches to be sufficiently small—as small as the value of p indicated by the above analysis—and compute sketches using stable distributions, then this can approximate the number of distinct items, and more generally the L_0 norm and L_0 difference between vectors. Since by definition the stable distributions capture the L_p norm, we have to take no special action when the vectors may contain negative values. Because the sketch is formed by a linear projection of random vectors with the input data, they naturally and smoothly accept updates of negative values.

When implementing this technique there are various technical details to deal with. Values drawn from stable distributions with small stability parameters α tend to grow very large, so even standard floating point formats are insufficient to handle them. However, in practice it usually suffices to set α to be a sufficiently small constant value. The experiments in [10] show that with $\alpha = 0.02$, good approximations to the L_0 norm and the number of distinct items can be found.

3.2 Dominance Norms

The approach of using stable distributions to capture the L_0 norm has been applied to other problems: in [13], the so-called “dominance norm” of data is approximated using stable distributions. The dominance norm is defined as $\sum_i \max_j a_{i,j}$ for a sequence of data items of the form $(i, a_{i,j})$, intuitively giving the “worst-case influence” of a sequence of signal values. This definition is illustrated in Fig. 3: for

the three signals shown on the left, the dominance norm is computed by finding the upper envelope of the signals (shown on the right), and taking the area under this upper envelope.

One can approximate this computation by transforming the input into an instance of computing L_0 norms. Suppose that the signal values $a_{i,j}$ are integers. We can (conceptually) replace each $a_{i,j}$ with a sequence of distinct items, $a_{i,1}, a_{i,2}, \dots, a_{i,j}$. Now observe that the number of distinct items in the transformed stream is exactly the dominance norm. This shows that L_0 is at the heart of the dominance norm. However, this approach is not scalable: naively replacing $a_{i,j}$ from the input with $a_{i,j}$ items means that the algorithm is exponentially slow in the size of the input. Instead, we can make use of the properties of stable distributions to build an estimator whose distribution is correct. The key idea is to round each $a_{i,j}$ to the closest power of $(1+\epsilon)$, $(1+\epsilon)^i$, say, and to add i appropriately scaled values from a stable distribution to build a sketch with the right distribution [13].

3.3 Application to Computing Embeddings

Not all objects can be naturally modeled as vectors. In dealing with massive items that consist of text, geometric data, structured data or other objects, new methods are needed to compare and measure them. However, the α -stable sketches for L_1 and L_2 distance are sufficiently flexible that they allow the following “embedding approach”. Consider any set of objects X , with a distance functions $D(q, r)$ defined for any $q, r \in X$.

Definition 4 A mapping $f : X \rightarrow L_p$ is called an *embedding with distortion c*, if for any $q, r \in X$, we have

$$D(q, r) \leq \|f(q) - f(r)\|_p \leq c \cdot D(q, r).$$

Here, we use L_p as shorthand for “a vector space with the vector L_p norm”. This definition can be further extended to allow the inequalities to hold with certain probability.

If the mapping f works for some $p \in \{0, 1, 2\}$, and if f can be computed in a streaming fashion, then we can obtain a streaming algorithm for computing short sketches of objects from the space X . That is, for any $q, r \in X$ defined by a stream, we can compute their sketches such that $D(q, r)$ can be approximated given the sketches. See [27, 33] for more on embeddings and their algorithmic applications.

The simplest example of this approach is given in [14], where it is shown that biologically motivated *distances on permutations* can be approximated up to small constant factors by encoding information about adjacent characters in the permutation as appropriate vectors in L_1 .

More involved is the method in [12] which shows that a *distance between strings* can also be embedded into L_1 . Only local information about the sequence is used in

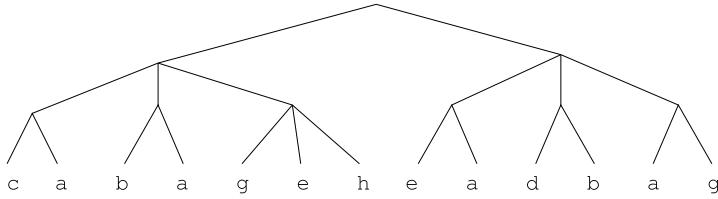


Fig. 4 Example parse tree for block edit distance text embedding

order to build the vector representation. The construction is more complex, since the sequence is parsed into small blocks, which in turn are re-parsed at successive levels in a hierarchy until a single item is left that represents the whole string. An example parsing of a string is shown in Fig. 4: a tree is built whose leaves are the characters of the string, and whose internal nodes represent selected substrings. The parsing can be computed as successive characters are observed, and the increasingly long substrings given by the internal nodes can be represented compactly with hash values. These substrings can be thought of as defining dimensions of a high-dimensional vector space. In the paper, it is shown that the L_1 distance between two vectors created by this process approximates an editing distance between the corresponding strings. Since the parsing can be computed online, sketches for this distance can be computed in small space using the α -stable approach. In total, $O(\log n \log^* n)$ space is required to process a string of length n , and the embedding has distortion $O(\log n \log^* n)$.

This approach is extended from string based data to tree structures (such as XML documents) in [22]. Using a similar parsing approach, it is shown how an appropriate editing distance on trees can be approximated up to a factor of $O(\log^2 n \log^* n)$ for trees with at most n nodes. Further, with a different kind of sketch based on stable distributions, the join size of a set of trees can be approximated. Here, the join size is the number of pairs that are within a threshold distance of each other.

Another example of this approach is given in [28]. Consider a discrete d -dimensional space $\{1, \dots, \Delta\}^d$, and let P and Q be two subsets from that space. Define $M(P, Q)$ to be the cost of the *matching* between P and Q with minimum cost: the cost of the matching is given by the sum of the distances between the paired-up points. The value of $M(P, Q)$ is a natural measure of a difference between two sets of points. Building on the work of Charikar [8], Indyk [28] showed that $M(\cdot, \cdot)$ can be embedded into L_1 with distortion $O(\log \Delta)$, and that embedding can be computed in small space. In fact, the embedding is quite simple. Let G_i , $i = 1, \dots, t = \log \Delta$, be square grids over \mathbb{R}^d with side length 2^{i-1} , shifted by a vector chosen uniformly at random from $[0, \Delta]^d$. For each cell c in G_i , let $n_P^i(c)$ be the number of points in P that fall into c ; note that n_P^i can be viewed as a (high-dimensional) vector. The embedding f maps P into (essentially) a concatenation of vectors $2^0 n_P^0, 2^1 n_P^1, \dots, 2^t n_P^t$. Observe that the embedding can be computed in a streaming fashion: adding a point p to P can be implemented by incrementing t positions in $f(P)$ that correspond to cells containing p ; deleting a point from P

can be implemented in an analogous way. Thus, the embedding can be naturally combined with the sketching algorithm from the previous section.

It is worth mentioning that the above approximation factor $O(\log \Delta)$ cannot be much improved if one insists on proceeding through the L_1 norm. Specifically, Naor and Schechtman [36] showed that any such embedding must incur $\mathcal{O}(\log \Delta)$ distortion. This lower bound may be tight for any approximation, and it will be interesting to resolve this issue.

Finally, we mention that an analogous embedding into L_0 gives a streaming algorithm for estimating the cost of the minimum spanning tree of set of points P , up to a factor of $O(\log \Delta)$; see [28] for details.

3.4 Clustering and Nearest Neighbors

The sketch structure can be used as a “distance oracle”, giving dependable approximations of the distance between high dimensional vectors while keeping only a constant amount of space for each object. They can therefore be applied to a number of data indexing and data mining questions which rely on such distance computations, replacing exact distance computations with approximations. For example, in order to perform clustering on a set of high dimensional vectors that are defined by data streams, we can keep sketches of the vectors, and then run the clustering algorithm using those sketches. This approach was investigated in [11], where experimental evidence was given that the clusterings found are of similar quality to those using exact distance measurements. The idea of replacing exact distance computations with approximate ones can be analyzed formally. For example, it is easy to show that for the k-center objective function that using approximate distances changes the approximation quality of the result from 2 to $2 + \epsilon$ [9].

A more involved approach was taken in [16]. This showed that sketches using stable distributions could be fitted into the framework of ‘Locality Sensitive Hash Functions’, and consequently can be used in the construction of Approximate Nearest Neighbor search structures. Although this more generally applies to non-streaming scenarios, the whole algorithm can be run on data presented in a streaming format. The space that is needed is a function of the number of data points, rather than a function of the total size of the input data.

4 Related Work

The sketch for L_2 , which is formed as the inner product between the vector a and vectors r , each of whose entries is drawn independently from a Gaussian distribution, can be seen as a weaker version of the well-known Johnson–Lindenstrauss Lemma [30]. This states that there exist embeddings of high dimensional vectors in Euclidean space into a space with dimension $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ which has distortion $1 + \epsilon$ with probability $1 - \delta$. Here, we have shown the result for a space where

we use the median operator to compute the distance. It has been shown that by taking the appropriately scaled L_2 difference between such sketch vectors formed in the same way also has this property (see, for example, [29]). What we also have here is a version of this “weak Johnson–Lindenstrauss” lemma for L_1 and L_0 . This is about as strong as we may hope for, since it has been shown that it is not possible to create an approximate distance preserving of L_1 into a lower dimensional L_1 space [4]. Here, we use an L_1 -like operator: instead of computing the L_1 norm of the sketch as $\sum_j |sk[j]|$, we compute $\text{med}_j(|sk[j]|)$.¹

The fundamental work of Alon, Matias and Szegedy [2] (described in an earlier chapter) initiated recent focus on computing norms of data streams. An algorithm given therein computes the second frequency moment of a data stream, F_2 . As was observed in [18], this directly gives a solution to finding the L_2 norm and L_2 difference between streams in the turnstile model. For most applications, the fact that updates can be performed very quickly, and that the necessary four-wise independent hash functions can be computed easily [39] means that this approach will be preferable in many situations.

For computing the L_1 difference, [18] shows how to modify the Alon–Matias–Szegedy method using carefully constructed range-summable random variables. However, this is under very strong restrictions on the data: each index can be seen at most once for each vector. The approach here allows a much more general model of the data, and is easier to compute. Similarly, [20] extended the above approach to arbitrary L_p norms for $p \in (0, 2)$, but with the same disadvantages. The main results described in this chapter on constructing sketches using stable distributions (Sect. 2) appeared first in [26].

The distinct elements problem has attracted a great deal of study. In the arrivals only (cash register) model, algorithms are known which are significantly faster than the approach described here; see [3, 17, 19, 23, 24], and the discussions in elsewhere in this book. In the more general problem of computing the L_0 norm and L_0 difference, where entries in the implicit vector defined by the stream can be negative, the method using stable distributions is the only published solution. A detailed empirical study of this approach, and a collection of ways to increase processing speed, are given in [10].

In terms of the application of α -stable sketches to speeding up clustering, see elsewhere in this book for details of much of the other work on clustering data streams. Typically, the goal is typically to compute a representation of the optimal clustering of a very large number of points in some arbitrary metric space, when each point has a small representation. Here, we considered a somewhat different scenario, where the number of points to cluster is not too large, but each point is represented by a very high dimensional vector in some L_p normed space. Hence, the two approaches are in some sense complementary and are not directly comparable.

¹Observe that since we need the median operator, this is not a normed space. This is an important restriction, since it means that one cannot immediately apply well-known techniques which work on specific normed spaces, such as clustering or similarity search. In contrast, since the Johnson–Lindenstrauss lemma does yield points in a lower dimensional metric space, all algorithms for Euclidean space can be applied to the resulting transformed data.

There is a very large body of work on Stable Distributions in Statistics and related areas. For pointers, see the books by Zolotarev [40, 42], and Nolan [38]. It is reasonable to say that the applications of stable distributions to streaming computations are far from exhausted.

5 Extensions and New Directions

Finally, we outline some potential areas for future research to extend the applications of stable distributions to streaming computations.

- One obstacle to implementing sketch-based summarization of very high speed data streams is that the time cost of maintaining sketches can be too expensive in some situations. This derives in part from the cost of simulating stable distributions using transforms from uniform distributions. The main cost comes from having to update every entry in the sketch with every update. For L_2 norms alternative methods are known which are asymptotically faster than $\Omega(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ per update; for example, see [7, 39] or the recent work on the “fast Johnson–Lindenstrauss transform” [1]. Likewise, for the problem of approximating the number of distinct items in the arrivals only (cash register) model, then faster updates are possible. It remains an open problem to design algorithms to compute L_1 norms and L_0 norms in the turnstile model, which have lower per-item update cost. Note that one cannot expect lower *space* costs, since lower bounds of $\Omega(\frac{1}{\epsilon^2})$ have been shown [41].
- It is of interest to address the engineering question of how to incorporate stable sketch computations into high speed data stream systems [15]. Various precomputations may be possible to speed up the computations, using appropriate look-up tables and so on. Techniques such as fixed point arithmetic may also be appropriate for certain fixed L_p values ($p = 1$ or $p = 2$, say), where the generated values do not grow too large. Other approaches may take advantages of skew in the data to, for example, collect together multiple instances of the same vector entry being updated, to further speed up the update time. There is a need to study in detail many implementation issues such as these to make the use of stable sketches within real situations a practical reality.
- The flexibility of this approach means that it is inviting to consider whether there are similar methods to compute other quantities of interest on the stream. For example, the “empirical entropy” of a sequence, given by $\sum_i a_i \log a_i$ has a number of applications, as does the “sum of logs”, $\sum_i \log a_i$. Recently, progress has been made on computing the empirical entropy of counts of items in the stream [5, 21], it remains open to determine whether stable distributions or similar techniques can also be applied to these problems.
- It is an intriguing fact that stable distributions exist only in the range $\alpha \in (0, 2]$, which corresponds to the range of L_p norms that can be approximated efficiently (essentially in constant space) on the stream. Meanwhile, there are provable lower bounds on the space required to estimate L_p norms for $p > 2$ that are polynomial

in n , the dimension of the vector. The connection between these facts may be more than mere coincidence, and making this connection explicit could lead to the development of stronger lower bounds, or lower bounds for other, related problems.

- Many techniques using stable distributions make use of a natural range summability-like property of these distributions. That is, their defining feature is that the sum of stable distributions is itself distributed stable. This results in careful constructions of random variables such that the range sum of particular subranges of variables can be computed efficiently (exponentially more efficient than directly computing the sum). Such constructions have been shown for $\alpha = 1$ and $\alpha = 2$ [25]. It remains to generalize these techniques to general values of α , and to show new applications.
- Finally, there is a large literature on stable distributions, resulting from their study in statistics, economics and beyond. Applications of stable distributions to streaming computations have only just begun to make use of the wealth of existing knowledge about these distributions, and it is very conceivable that there are many other applications of these distributions to problems of practical interest in streaming computations.

References

1. N. Ailon, B. Chazelle, Approximate nearest neighbors and the fast Johnson–Lindenstrauss transform, in *Proceedings of the ACM Symposium on Theory of Computing* (2006)
2. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in *Proceedings of the ACM Symposium on Theory of Computing* (1996), pp. 20–29. Journal version in *J. Comput. Syst. Sci.* **58**, 137–147 (1999)
3. Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, Counting distinct elements in a data stream, in *Proceedings of RANDOM 2002* (2002), pp. 1–10
4. B. Brinkman, M. Charikar, On the impossibility of dimensionality reduction in L_1 , in *IEEE Conference on Foundations of Computer Science* (2003), pp. 514–523
5. A. Chakrabarti, G. Cormode, A. McGregor, A near-optimal algorithm for computing the entropy of a stream, in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms* (2007)
6. J.M. Chambers, C.L. Mallows, B.W. Stuck, A method for simulating stable random variables. *J. Am. Stat. Assoc.* **71**(354), 340–344 (1976)
7. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)* (2002), pp. 693–703
8. M.S. Charikar, Similarity estimation techniques from rounding algorithms, in *Proceedings of the ACM Symposium on Theory of Computing* (2002), pp. 380–388
9. G. Cormode, Sequence distance embeddings. PhD thesis, University of Warwick (2003)
10. G. Cormode, M. Datar, P. Indyk, S. Muthukrishnan, Comparing data streams using Hamming norms, in *Proceedings of the International Conference on Very Large Data Bases* (2002), pp. 335–345. Journal version in *IEEE Trans. Knowl. Data Eng.* **15**(3), 529–541 (2003)
11. G. Cormode, P. Indyk, N. Koudas, S. Muthukrishnan, Fast mining of tabular data via approximate distance computations, in *Proceedings of the International Conference on Data Engineering* (2002), pp. 605–616
12. G. Cormode, S. Muthukrishnan, The string edit distance matching problem with moves, in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms* (2002), pp. 667–676

13. G. Cormode, S. Muthukrishnan, Estimating dominance norms of multiple data streams, in *Proceedings of the European Symposium on Algorithms (ESA)*. LNCS, vol. 2838 (2003)
14. G. Cormode, S. Muthukrishnan, S.C. Sahinalp, Permutation editing and matching via embeddings, in *Proceedings of 28th International Colloquium on Automata, Languages and Programming*, vol. 2076 (2001), pp. 481–492
15. C. Cranor, T. Johnson, O. Spatscheck, V. Shkapenyuk, Gigascope: a stream database for network applications, in *Proceedings of ACM SIGMOD International Conference on Management of Data* (2003), pp. 647–651
16. M. Datar, N. Immorlica, P. Indyk, V.S. Mirrokni, Locality-sensitive hashing scheme based on p -stable distributions, in *Symposium on Computational Geometry* (2004)
17. C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links, in *Proceedings of the Internet Measurement Conference* (2003), pp. 153–166
18. J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, An approximate L_1 -difference algorithm for massive data streams, in *IEEE Conference on Foundations of Computer Science* (1999), pp. 501–511
19. P. Flajolet, G.N. Martin, Probabilistic counting, in *IEEE Conference on Foundations of Computer Science* (1983), pp. 76–82. Journal version in *J. Comput. Syst. Sci.* **31**, 182–209 (1985)
20. J. Fong, M. Strauss, An approximate L_p -difference algorithm for massive data streams, in *Symposium on Theoretical Aspects of Computer Science (STACS)* (2000), pp. 193–204
21. S. Ganguly, B. Lakshminath, Estimating entropy over data streams, in *Proceedings of the European Symposium on Algorithms (ESA)* (2006)
22. M. Garofalakis, A. Kumar, Correlating XML data streams using tree-edit distance embeddings, in *Proceedings of ACM Principles of Database Systems* (2003), pp. 143–154
23. P. Gibbons, Distinct sampling for highly-accurate answers to distinct values queries and event reports, in *Proceedings of the International Conference on Very Large Data Bases* (2001), pp. 541–550
24. P. Gibbons, S. Tirthapura, Estimating simple functions on the union of data streams, in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2001), pp. 281–290
25. A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M. Strauss, Fast, small-space algorithms for approximate histogram maintenance, in *Proceedings of the ACM Symposium on Theory of Computing* (2002), pp. 389–398
26. P. Indyk, Stable distributions, pseudorandom generators, embeddings and data stream computation, in *IEEE Conference on Foundations of Computer Science* (2000), pp. 189–197
27. P. Indyk, Algorithmic aspects of geometric embeddings (invited tutorial), in *IEEE Conference on Foundations of Computer Science* (2001), pp. 10–35
28. P. Indyk, Algorithms for dynamic geometric problems over data streams, in *Proceedings of the ACM Symposium on Theory of Computing* (2004)
29. P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in *Proceedings of the ACM Symposium on Theory of Computing* (1998), pp. 604–613
30. W.B. Johnson, J. Lindenstrauss, Extensions of Lipschitz mapping into Hilbert space. *Contemp. Math.* **26**, 189–206 (1984)
31. P. Li, Very sparse stable random projections, estimators and tail bounds for stable random projections. Technical report (2006). [arXiv:cs/0611114](https://arxiv.org/abs/cs/0611114)
32. P. Li, T. Hastie, K.W. Church, Nonlinear estimators and tail bounds for dimension reduction in L_1 using Cauchy random projections. *J. Mach. Learn. Res.* (2007)
33. J. Matoušek, *Lectures on Discrete Geometry* (Springer, Berlin, 2002)
34. R. Motwani, P. Raghavan, *Randomized Algorithms* (Cambridge University Press, Cambridge, 1995)
35. S. Muthukrishnan, Data streams: algorithms and applications, in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms* (2003)
36. A. Naor, G. Schechtman, Planar earthmover is not in L_1 , in *IEEE Conference on Foundations of Computer Science* (2006)

37. N. Nisan, Pseudorandom generators for space-bounded computation. *Combinatorica* **12**, 449–461 (1992)
38. J. Nolan, Stable distributions. Available from <http://academic2.american.edu/~jpnnolan/stable/chap1.ps>
39. M. Thorup, Y. Zhang, Tabulation based 4-universal hashing with applications to second moment estimation, in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms* (2004), pp. 615–624
40. V.V. Uchaikin, V.M. Zolotarev, *Chance and Stability: Stable Distributions and Their Applications* (VSP, Utrecht, 1999)
41. D. Woodruff, Optimal space lower bounds for all frequency moments, in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms* (2004), pp. 167–175
42. V.M. Zolotarev, *One Dimensional Stable Distributions*. Translations of Mathematical Monographs, vol. 65 (Am. Math. Soc., Providence, 1983)

Tracking Queries over Distributed Streams

Minos Garofalakis

Effective Big Data analytics pose several difficult challenges for modern data management architectures. One key such challenge arises from the naturally streaming nature of big data, which mandates efficient algorithms for querying and analyzing massive, continuous data streams (that is, data that is seen only once and in a fixed order) with limited memory and CPU-time resources. Such streams arise naturally in emerging large-scale event monitoring applications; for instance, network-operations monitoring in large ISPs where usage information from numerous sites needs to be continuously collected and analyzed for interesting trends. In addition to memory- and time-efficiency concerns, the inherently distributed nature of such applications also raises important communication-efficiency issues, making it critical to carefully optimize the use of the underlying network infrastructure. In this chapter, we provide a brief introduction to the distributed data streaming model and the Geometric Method (GM), a generic technique for effectively tracking complex queries over massive distributed streams. We also discuss several recently-proposed extensions to the basic GM framework, such as the combination with stream-sketching tools and local prediction models, as well as more recent developments leading to a more general theory of Safe Zones and interesting connections to convex Euclidean geometry. Finally, we outline various challenging directions for future research in this area.

M. Garofalakis (✉)

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

1 Introduction

Traditional data-management systems are typically built on a *pull-based paradigm*, where users issue one-shot queries to static data sets residing on disk, and the system processes these queries and returns their results. For several emerging application domains, however, data arrives and needs to be processed on a continuous (24×7) basis, without the benefit of several passes over a static, persistent data image. These *continuous data streams* arise naturally in new large-scale event monitoring applications that require the ability to efficiently process continuous, high-volume streams of data in real time. Such monitoring systems are routinely employed, for instance, 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 examples include real-time analysis tools for financial data streams, and event and operations monitoring applications for enterprise clouds and data centers. As both the scale of today's networked systems and the volumes and rates of the associated data streams continue to increase with no bound in sight, algorithms and tools for effectively analyzing them are becoming an important research mandate.

Large-scale stream processing applications rely on *continuous*, event-driven monitoring, that is, real-time tracking of measurements and events, rather than one-shot answers to sporadic queries. Furthermore, the vast majority of these applications are inherently *distributed*, with several remote monitor sites observing their local, high-speed data streams and exchanging information through a communication network. This distribution of the data naturally implies critical communication constraints that typically prohibit centralizing all the streaming data, due to either the huge volume of the data (e.g., in IP-network monitoring, where the massive amounts of collected utilization and traffic information can overwhelm the production IP network [13]), or power and bandwidth restrictions (e.g., in wireless sensornets, where communication is the key determinant of sensor battery life [27]). Finally, an important requirement of large-scale event monitoring is the effective support for tracking complex, *holistic queries* that provide a global view of the data by combining and correlating information across the collection of remote monitor sites. For instance, tracking aggregates over the result of a distributed join (the “workhorse” operator for combining tables in relational databases) can provide unique, real-time insights into the workings of a large-scale distributed system, including system-wide correlations and potential anomalies [7]. Monitoring the precise value of such holistic queries without continuously centralizing all the data seems hopeless; luckily, when tracking statistical behavior and patterns in large scale systems, *approximate answers* (with reasonable approximation error guarantees) are often sufficient. This often allows algorithms to effectively tradeoff efficiency with approximation quality (e.g., using sketch-based stream approximations [7]).

Given the prohibitive cost of data centralization, it is clear that realizing sophisticated, large-scale distributed data-stream analysis tools must rely on novel

algorithmic paradigms for processing local streams of data *in situ* (i.e., locally at the sites where the data is observed). This, of course, implies the need for intelligently decomposing a (possibly complex) global data-analysis and monitoring query into a collection of “safe” local queries that can be tracked independently at each site (without communication), while guaranteeing correctness for the global monitoring operation. This decomposition process can enable truly distributed, event-driven processing of real-time streaming data, using a *push-based paradigm*, where sites monitor their local queries and communicate only when some local query constraints are violated [7, 33]. Nevertheless, effectively decomposing a complex, holistic query over the global collections of streams into such local constraints is far from straightforward, especially in the case of *non-linear* queries (e.g., norms or joins) [33].

The bulk of early work on data-stream processing has focused on developing space-efficient, one-pass algorithms for performing a wide range of *centralized computations* on massive data streams; examples include computing quantiles [21], estimating distinct values [20], and set-expression cardinalities [16], counting frequent elements (i.e., “heavy hitters”) [5, 11, 28], approximating large Haar-wavelet coefficients [10], and estimating join sizes and stream norms [1, 2, 15]. Monitoring *distributed* data streams has attracted substantial research interest in recent years [6, 30], with early work focusing on the monitoring of *single values*, and building appropriate models and filters to avoid propagating updates if these are insignificant compared to the value of simple *linear* aggregates (e.g., to the SUM of the distributed values). For instance, [31] proposes a scheme based on “adaptive filters,” that is, bounds around the value of distributed variables, which shrink or grow in response to relative stability or variability, while ensuring that the total uncertainty in the bounds is at most a user-specified bound. Still, in the case of linear aggregate functions, deriving local filter bounds based on a global monitoring condition is rather straightforward, with the key issue being how to intelligently distribute the available aggregate “slack” across all sites [3, 9, 23].

In this chapter, we focus on recently-developed algorithmic tools for effectively tracking a broad class of complex queries over massive, distributed data streams. We start by describing the key elements of a generic *distributed stream-processing model* and define a broad class of distributed query-tracking problems addressed by our techniques. We then give an overview of the *Geometric Method (GM)* [24, 33] for distributed threshold monitoring that lies at the core of our distributed query-tracking methodology, and briefly discuss recent extensions to the basic GM framework that incorporate stream sketches [17] and local prediction models [18, 19]. We also summarize recent developments leading to a more general theory of *Safe Zones* for geometric monitoring and interesting connections to convex Euclidean geometry [26]. Finally, we conclude with a brief discussion of new research directions in this space.

2 Distributed Data Streaming and the Geometric Method

In this section, we discuss some key concepts behind data-stream processing and the distributed streaming model. We then introduce the Geometric Method (GM) [33], a generic technique for query monitoring over distributed streams.

2.1 Data Streams and Distributed Streaming

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). Such stream queries are typically *continuous*, implying the need for continuous, real-time monitoring of the query answer over the changing stream.

As discussed earlier in this volume, a data stream can be modeled as a *massive, dynamic, one-dimensional vector* $\mathbf{v}[1 \dots N]$ that, at any point in time, captures the current state of the stream. (Multi-dimensional data can naturally be handled in this abstract model by simply “unfolding” the corresponding multi-dimensional frequency distribution on one vector dimension using standard techniques, e.g., row- or column-major). The dynamic vector \mathbf{v} is rendered through a continuous stream of updates, where each update effectively modifies values in \mathbf{v} —the nature of these update operations gives rise to different data streaming models, such as *time-series*, *cash-register*, and *turnstile* streams [29].

Data-stream processing algorithms aim to compute functions (or, queries) on the stream vector \mathbf{v} at different points during the lifetime of the stream (continuous or ad-hoc). Since N can be very large, the typical requirement here is that these algorithms work in *small space* (i.e., the state maintained by the algorithm) and *small time* (i.e., the processing time per update), where “small” is understood to mean a quantity significantly smaller than $\Theta(N)$ (typically, poly-logarithmic in N). Several such stream-processing algorithms are known for various data-analysis queries [1, 2, 5, 10, 11, 15, 16, 20, 21, 28].

The naturally distributed nature of large-scale event-monitoring applications implies one additional level of complexity, in the sense that there is no centralized observation point for the dynamic stream vector \mathbf{v} ; instead, \mathbf{v} is distributed across several sites. More specifically, we consider a distributed computing environment, comprising a collection of k *remote sites* and a designated *coordinator site*. Streams of data updates arrive continuously at remote sites, while the coordinator site is responsible for generating approximate answers to (possibly, continuous) user queries posed over the collection of remotely-observed streams (across all sites). Following earlier work in the area [3, 7, 9, 14, 31], our distributed stream-processing model does not explicitly allow direct communication between remote sites; instead, as illustrated in Fig. 1(a), a remote site exchanges messages only with the coordinator,

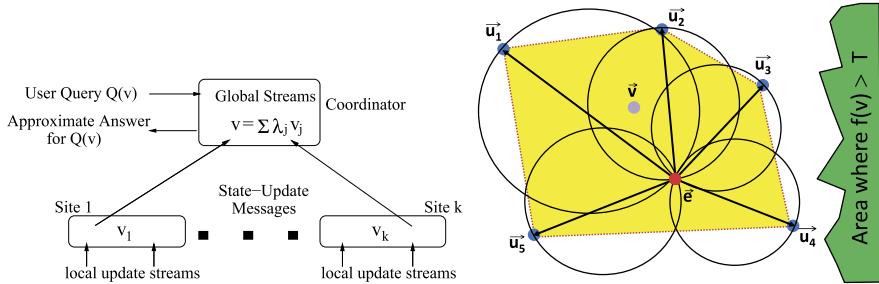


Fig. 1 (a) Distributed stream processing architecture. (b) Geometric method: estimate vector e , drift vectors \mathbf{u}_j , convex hull enclosing current \mathbf{v} (dotted outline), and bounding balls $B(\mathbf{e} + \frac{1}{2} \Delta \mathbf{v}_j, \frac{1}{2} \|\Delta \mathbf{v}_j\|)$

providing it with state information on its (locally-observed) streams.¹ Note that such a hierarchical processing model is, in fact, representative of a large class of applications, including network monitoring where a central Network Operations Center (NOC) is responsible for processing network traffic statistics (e.g., link bandwidth utilization, IP source–destination byte counts) collected at switches, routers, and/or Element Management Systems (EMSs) distributed across the network.

Each remote site $j \in \{1, \dots, k\}$ observes (possibly, several) local update streams that incrementally render a *local stream vector* \mathbf{v}_j capturing the current local state of the observed stream(s) at site j . All local stream vectors \mathbf{v}_j in our distributed streaming architecture change dynamically over time—when necessary, we make this dependence explicit, using $\mathbf{v}_j(t)$ to denote the state of the vector at time t (assuming a consistent notion of “global time” in our distributed system). The unqualified notation \mathbf{v}_j typically refers to the *current* state of the local stream vector.

We define the *global stream vector* \mathbf{v} of our distributed stream(s) as any *weighted average* (i.e., convex combination) of the local stream vectors $\{\mathbf{v}_j\}$, that is, $\mathbf{v} = \sum_{j=1}^k \lambda_j \mathbf{v}_j$, where $\sum_j \lambda_j = 1$ and $\lambda_j \geq 0$ for all j . (Again, to simplify notation, we typically omit the explicit dependence on time when referring to the current global vector.) Our focus is on the problem of effectively answering user queries (or, functions) over the global stream vector at the coordinator site. Rather than one-time query/function evaluation, we assume a continuous-querying environment which implies that the coordinator needs to *continuously maintain* (or, *track*) the answers to queries as the local update streams \mathbf{v}_j evolve at individual remote sites. There are two defining characteristics of our problem setup that raise difficult algorithmic challenges for our query tracking problems:

- *The distributed nature and large volumes of local streaming data* raise important communication and space/time efficiency concerns. Naïve schemes that accurately track query answers by forcing remote sites to ship every remote stream update to

¹Of course, sites can always communicate with each other through the coordinator—this would only increase communication load by a factor of 2.

the coordinator are clearly impractical, since they can impose an inordinate burden on the underlying communication infrastructure (especially, for high-rate data streams and large numbers of remote sites). Furthermore, the voluminous nature of the local data streams implies that effective streaming tools are needed at the remote sites in order to manage the local stream vectors in sublinear space/time. Thus, a practical approach is to adopt the paradigm of continuous tracking of *approximate* query answers at the coordinator site with strong guarantees on the quality of the approximation. This allows schemes that can effectively trade-off space/time/communication efficiency and query-approximation accuracy in a precise, quantitative manner.

- *General, non-linear queries/functions* imply fundamental and difficult challenges for distributed monitoring. For the case of linear functions, a number of approaches have been proposed that rely on the key idea of allocating appropriate “*slack*s” to the remote sites based on their locally-observed function values (e.g., [3, 23, 31]). Unfortunately, it is not difficult to find examples of simple *non-linear* functions on one-dimensional data, where it is basically impossible to make any assumptions about the value of the global function based on the values observed locally at the sites [33]. This renders conventional slack-allocation schemes inapplicable in this more general setting.

2.2 The Geometric Method (GM)

Sharfman et al. [33] consider the fundamental problem of *distributed threshold monitoring*, that is, determining whether $f(\mathbf{v}) < \tau$ or $f(\mathbf{v}) > \tau$, for a given (general) function $f()$ over the global stream vector and a fixed threshold τ . Their key idea is that, since it is generally impossible to connect the locally-observed values of $f()$ to the global value $f(\mathbf{v})$, one can employ geometric arguments to monitor the *domain* (rather than the range) of the monitored function $f()$. More specifically, assume that at any point in time, each site j has informed the coordinator of some prior state of its local vector \mathbf{v}_j^p ; thus, the coordinator has an estimated global vector $\mathbf{e} = \mathbf{v}^p = \sum_{j=1}^k \lambda_j \mathbf{v}_j^p$. Clearly, the updates arriving at sites can cause the local vectors \mathbf{v}_j to drift too far from their previously reported values \mathbf{v}_j^p , possibly leading to a violation of the τ threshold. Let $\Delta\mathbf{v}_j = \mathbf{v}_j - \mathbf{v}_j^p$ denote the local *delta vector* (due to updates) at site j , and let $\mathbf{u}_j = \mathbf{e} + \Delta\mathbf{v}_j$ be the *drift vector* from the previously reported estimate at site j . We can then express the current global stream vector \mathbf{v} in terms of the drift vectors:

$$\mathbf{v} = \sum_{j=1}^k \lambda_j (\mathbf{v}_j^p + \Delta\mathbf{v}_j) = \mathbf{e} + \sum_{j=1}^k \lambda_j \Delta\mathbf{v}_j = \sum_{j=1}^k \lambda_j (\mathbf{e} + \Delta\mathbf{v}_j).$$

That is, the current global vector is a convex combination of drift vectors and, thus, is guaranteed to lie somewhere within the convex hull of the delta vectors around \mathbf{e} .

Figure 1(b) depicts an example in $d = 2$ dimensions. The current value of the global stream vector lies somewhere within the shaded convex-hull region; thus, as long as the convex hull does not overlap the inadmissible region (i.e., the region $\{\mathbf{v} \in \mathbb{R}^2 : f(\mathbf{v}) > \tau\}$ in Fig. 1(b)), we can guarantee that the threshold has not been violated (i.e., $f(\mathbf{v}) \leq \tau$).

The problem, of course, is that the $\Delta\mathbf{v}_j$'s are spread across the sites and, thus, the above condition cannot be checked locally. To transform the global condition into a local constraint, we place a d -dimensional *bounding ball* $B(\mathbf{c}, r)$ around each local delta vector, of radius $r = \frac{1}{2}\|\Delta\mathbf{v}_j\|$ and centered at $\mathbf{c} = \mathbf{e} + \frac{1}{2}\Delta\mathbf{v}_j$ (see Fig. 1(b)). It can be shown that the union of all these balls completely covers the convex hull of the drift vectors [33]. This observation effectively reduces the problem of monitoring the global stream vector to the local problem of each remote site monitoring the ball around its local delta vector.

More specifically, given the monitored function $f()$ and threshold τ , we can partition the d -dimensional space into two sets $A = \{\mathbf{v} : f(\mathbf{v}) \leq \tau\}$ and $\overline{A} = \{\mathbf{v} : f(\mathbf{v}) > \tau\}$. (Note that these sets can be arbitrarily complex, e.g., they may comprise multiple disjoint regions of \mathbb{R}^d .) The basic protocol is now quite simple: Each site monitors its delta vector $\Delta\mathbf{v}_j$ and, with each update, checks whether its bounding ball $B(\mathbf{e} + \frac{1}{2}\Delta\mathbf{v}_j, \frac{1}{2}\|\Delta\mathbf{v}_j\|)$ is *monochromatic*, i.e., all points in the ball lie within the same region (A or \overline{A}). If this is not the case, we have a *local threshold violation*, and the site communicates its local $\Delta\mathbf{v}_j$ to the coordinator. The coordinator then initiates a *synchronization process* that typically tries to resolve the local violation by communicating with only a subset of the sites in order to “balance out” the violating $\Delta\mathbf{v}_j$ and ensure the monochromicity of all local bounding balls [33]. In the worst case, the delta vectors from all k sites are collected, leading to an accurate estimate of the current global stream vector, which is by definition monochromatic (since all bounding balls have 0 radius).

The power of the GM stems from the fact that it is essentially agnostic of the specific (global) function $f(\mathbf{v})$ being monitored.² Note that the function itself is only used at a remote site when checking the monochromicity of its local ball, which essentially boils down to solving a minimization/maximization problem for $f()$ within the area of that ball. This may, of course, be complex but it also enables the GM to effectively trade local computation for communication.

From Threshold Crossing to Approximate Query Tracking

Consider the task of monitoring (at the coordinator) the value of a function $f()$ over the global stream vector \mathbf{v} to within θ relative error. (Our discussion here focuses on relative error—the case of monitoring to within bounded *absolute* error can be

²The assumption that GM only monitors functions of the (weighted) average of local stream vectors is not really restrictive: Numerous complex functions can actually be expressed as functions of the average using simple tricks, such as adding additional dimensions to the stream vectors; see, e.g., [4].

handled in a similar manner.) Since all the coordinator has is the estimated value of the global stream vector $\mathbf{e} = \mathbf{v}^P$ based on the most recent site updates v_j^P , our monitoring protocol would have to guarantee that the estimated function value carries at most θ relative error compared to the up-to-date value $f(\mathbf{v}) = f(\mathbf{v}(t))$, that is, $f(\mathbf{v}^P) \in (1 \pm \theta) f(\mathbf{v})$,³ which is obviously equivalent to monitoring two threshold queries on $f(\mathbf{v})$:

$$f(\mathbf{v}) \geq \frac{f(\mathbf{v}^P)}{1 + \theta} \quad \text{and} \quad f(\mathbf{v}) \leq \frac{f(\mathbf{v}^P)}{1 - \theta}.$$

These are exactly the threshold conditions that our approximate function tracking protocols will need to monitor. Note that $f(\mathbf{v}^P)$ in the above expression is a constant (based on the latest communication of the coordinator with the remote sites). Similar threshold conditions can also be derived when the local/global values of $f()$ are only known approximately (e.g., using sketches [1, 2] or other streaming approximations)—the threshold conditions just need to account for the added approximation error [17].

3 Enhancing GM: Sketches and Prediction Models

In this section, we give a brief overview of more recent work on extending the GM with two key stream-processing tools, namely sketches and prediction models [17–19].

3.1 GM and AMS Sketches

Techniques based on small-space pseudo-random *sketch* summaries of the data have proved to be very effective tools for dealing with massive, rapid-rate data streams in centralized settings [8]. The key idea in such sketching techniques is to represent a streaming frequency vector \mathbf{v} using a much smaller (typically, randomized) *sketch* vector (denoted by $\text{sk}(\mathbf{v})$) that (i) can be easily maintained as the updates incrementally rendering \mathbf{v} are streaming by, and (ii) provide probabilistic guarantees for the quality of the data approximation. The widely used AMS sketch (proposed by Alon, Matias, and Szegedy in their seminal paper [2], also discussed in detail earlier in this volume) defines the entries of the sketch vector as *pseudo-random linear projections* of \mathbf{v} that can easily be maintained over the stream of updates. More specifically, each entry of $\text{sk}(\mathbf{v})$ is essentially a projection (i.e., an inner product) of the \mathbf{v} vector over a family of 4-wise independent pseudo-random variates that can be easily maintained (using a simple counter) over the input update stream. Another important property is the *linearity* of AMS sketches: Given two “parallel” sketches

³Throughout, the notation $x \in (y \pm z)$ is equivalent to $|x - y| \leq |z|$.

(built using the same pseudo-random families) $\text{sk}(\mathbf{v}_1)$ and $\text{sk}(\mathbf{v}_2)$, the sketch of the union of the two underlying streams (i.e., the streaming vector $\mathbf{v}_1 + \mathbf{v}_2$) is simply the component-wise sum of their sketches, that is, $\text{sk}(\mathbf{v}_1 + \mathbf{v}_2) = \text{sk}(\mathbf{v}_1) + \text{sk}(\mathbf{v}_2)$. This linearity makes such sketches particularly useful in distributed settings.

AMS sketch estimators can effectively approximate *inner-product queries* $\mathbf{v} \cdot \mathbf{u} = \sum_i \mathbf{v}[i] \cdot \mathbf{u}[i]$ over streaming data vectors and tensors. As discussed earlier in this volume, such inner products naturally map to several interesting query classes (e.g., join and multi-join aggregates, range and quantile queries, heavy hitters and top- k queries, and approximate histogram and wavelet representations). The AMS estimator function $f_{\text{AMS}}()$ computed over the sketch vectors of \mathbf{v} and \mathbf{u} is complex, and involves both averaging and median-selection over the components of the sketch-vector inner product [1, 2]. Formally, viewing each sketch vector as a two-dimensional $n \times m$ matrix (where $n = O(\frac{1}{\epsilon^2})$, $m = O(\log(1/\delta))$) and $\epsilon, 1 - \delta$ denote desired bounds on error and probabilistic confidence (respectively)), the AMS estimator function is defined as

$$f_{\text{AMS}}(\text{sk}(\mathbf{v}), \text{sk}(\mathbf{u})) = \underset{i=1, \dots, m}{\text{median}} \left\{ \frac{1}{n} \sum_{l=1}^n \text{sk}(\mathbf{v})[l, i] \cdot \text{sk}(\mathbf{u})[l, i] \right\}, \quad (1)$$

and guarantees that, with probability $\geq 1 - \delta$, $f_{\text{AMS}}(\text{sk}(\mathbf{v}), \text{sk}(\mathbf{u})) \in (\mathbf{v} \cdot \mathbf{u} \pm \epsilon \|\mathbf{v}\| \|\mathbf{u}\|)$ [1, 2].

Moving to the distributed streams setting, note that our discussion of the GM thus far has assumed that all remote sites maintain the *full stream vector* (i.e., employ $\Theta(N)$ space), which is often unrealistic for real-life data streams. In our recent work [17], we have proposed novel approximate query tracking protocols that exploit the combination of the GM and AMS sketch estimators. The AMS sketching idea offers an effective streaming dimensionality-reduction tool that significantly expands the scope of the original GM, allowing it to handle massive, high-dimensional distributed data streams in an efficient manner with approximation-quality guarantees. A key technical observation is that, by exploiting properties of the AMS estimator function, geometric monitoring can now take place in a *much lower-dimensional space*, allowing for communication-efficient monitoring. In fact, we show that, using appropriate lower and upper bounds on $f_{\text{AMS}}()$, we can monitor a function in only m -dimensional space (where $m = O(\log(1/\delta))$). This is a crucial optimization since sketch matrices are typically very “thin”, i.e., $n \gg m$, as n depends quadratically on the sketching error ϵ , whereas m depends only logarithmically on the desired confidence δ . Another technical challenge that arises is how to effectively test the monochromicity of bounding balls in this lower-dimensional space with respect to threshold conditions involving the highly non-linear median operator in the AMS estimator $f_{\text{AMS}}()$ (see Eq. (1)). We have proposed a number of novel algorithmic techniques to address the technical challenges that arise, starting from the easier cases of L_2 -norm (i.e., self-join) and range queries, and then extending them to the case of general inner-product (i.e., binary-join) queries. Our experimental study with real-life data sets demonstrates the practical benefits of our approach, showing consistent gains in communication cost compared to state-of-the-art methods.

3.2 GM and Prediction Models

In other recent work [18, 19], we have proposed a novel combination of the geometric method with *local prediction models* for describing the temporal evolution of local data streams. (The adoption of prediction models has already been proven beneficial in terms of bandwidth preservation in distributed settings [7].) We demonstrate that prediction models can be incorporated in a very natural way in the geometric method for tracking general, non-linear functions; furthermore, we show that the initial geometric monitoring method of Sharfman et al. [24, 33] is only a special case of our, more general, prediction-based geometric monitoring framework. Interestingly, the mere utilization of local predictions is not enough to guarantee lower communication overheads even when predictors are quite capable of describing local stream distributions. We establish a theoretically solid monitoring framework that incorporates conditions that can lead to fewer contacts with the coordinator. We also develop a number of mechanisms, along with extensive probabilistic models and analysis, that relax the previously introduced framework, base their function on simpler criteria, and yield significant communication benefits in practical scenarios.

4 Towards Convex Safe Zones

In followup work to the GM, Keren et al. [24] propose a simple, generic geometric monitoring strategy that can be formally shown to encompass the original GM scheme as a special case. Briefly, assuming we are monitoring the threshold condition $f(\mathbf{v}) \leq \tau$, the idea is to define a certain *convex subset* C of the admissible region $A = \{\mathbf{v} : f(\mathbf{v}) \leq \tau\}$ (i.e., a *convex admissible subset*), which is then used to define *Safe Zones* (SZs) for the local drift vectors: *Site j simply monitors the condition $\mathbf{u}_j = \mathbf{e} + \Delta\mathbf{v}_j \in C$.* The correctness of this generic monitoring scheme follows directly from the convexity of C , and our earlier observation that the global stream vector \mathbf{v} always lies in the convex hull of \mathbf{u}_j , $j = 1, \dots, k$: If $\mathbf{u}_j \in C$ for all nodes j then, by convexity, this convex hull (and, therefore \mathbf{v}) lies completely within C and, therefore, the admissible region (since $C \subseteq A$). (Note that the convexity of C plays a crucial role in the above correctness argument.)

While the convexity of C is needed for the *correctness* of the monitoring scheme, it is clear that the size of C plays a critical role in its *efficiency*: Obviously, a larger C implies fewer local violations and, thus, smaller communication/synchronization overheads. This, in turn, implies a fairly obvious dominance relationship over geometric distributed monitoring schemes: Given two geometric algorithms \mathcal{A}_1 and \mathcal{A}_2 (for the same distributed monitoring problem) that use the convex admissible subsets C_1 and C_2 (respectively), algorithm \mathcal{A}_1 is *provably superior* to \mathcal{A}_2 if $C_2 \subset C_1$. Note that, in the simple case of *linear* functions $f()$, the admissible region A itself is convex, and therefore one can choose $C = A$; however, for more complicated, non-linear functions, A is non-convex and quite complex. Thus, finding a “large” convex subset of A is a crucial component of effective geometric monitoring. This

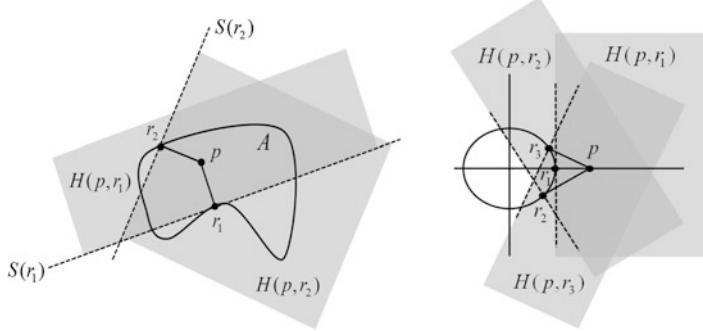


Fig. 2 (a) An equivalent construction of C_{GM} by intersecting half-spaces (shaded regions), depicted for two points, r_1, r_2 on A 's boundary. C_{GM} is obtained by intersecting all such half-spaces. (b) Construction of C_{GM} when \bar{A} is the unit disk. Three of the half-spaces whose intersection equals C_{GM} are depicted

is, of course, a very difficult optimization problem, in general; furthermore, unlike the GM's generic “bounding ball” geometric constraints, finding an effective convex admissible subset depends very much on the specific function $f()$ being monitored.

Interestingly, the bounding ball constraints of the GM can also be cast in terms of a convex admissible subset (denoted by C_{GM}) that can be mathematically shown to be equivalent to the intersection of the (possibly, infinitely many) half-spaces supported by hyperplanes defined by the points at the boundary of the admissible region A (see Fig. 2(a)) [24]. As demonstrated in our recent work [26], while the GM can achieve good results and is generic (i.e., can be directly applied to any monitoring function), its performance can be far from optimal since its underlying SZ C_{GM} is often far too restrictive. In several practical scenarios, C_{GM} can be drastically improved by intersecting *much fewer half-spaces* in order to obtain provably larger convex admissible subsets, giving significantly more efficient monitoring schemes. As a simple example, consider the scenario depicted in Fig. 2(b): The monitored function is $f(\mathbf{v}) = \|\mathbf{v}\|^2 \geq 1$, which implies that the inadmissible region \bar{A} is the unit disk. Clearly, while the C_{GM} defined by the intersection of the half-spaces for all r_i 's on the boundary of the unit disk is a correct SZ, it is also far from optimal: It is easy to see that the (single) half-space $H(p, r_1)$ supported by the hyperplane through r_1 is a better SZ, as it satisfies the necessary convexity property and strictly contains C_{GM} . In this case, just a single supporting hyperplane suffices to separate the entire inadmissible region \bar{A} from the current estimate/reference point p and, thus, there is no need to further intersect with other half-spaces ($H(p, r_2), H(p, r_3)$, and so on).

In a nutshell, our proposed *Convex Decomposition* (CD) method [26] generalizes the above observation—it works by identifying convex subsets of the *inadmissible region*, and using them to define non-redundant collections of half-spaces that separate these subsets from the admissible region. Our CD methodology can be applied to several important approximate query monitoring tasks (e.g., norms, range aggregates, and joins) giving provably larger SZs and substantially better performance

than the original GM. Of course, the general problem of determining effective SZs for arbitrary functions remains open.

5 Conclusions and Future Directions

We have given a brief introduction to the distributed data streaming model and the Geometric Method (GM), a generic technique for effectively tracking complex queries over massive distributed streams. We have also discussed recently-proposed extensions to the basic GM framework, such as the combination with AMS stream sketches and local prediction models, as well as recent developments leading to a more general theory of *Safe Zones* for geometric monitoring and interesting connections to convex Euclidean geometry. The GM framework provides a very powerful tool for dealing with continuous query computations over distributed streaming data; see, for instance, [32] for a novel application of the GM to continuous monitoring of skyline queries over fragmented dynamic data.

Continuous distributed streaming is a vibrant, rapidly evolving field of research, and a community of researchers has started forming around theoretical, algorithmic, and systems issues in the area [30]. Naturally, there are several promising directions for future research. First, the single-level hierarchy model (depicted in Fig. 1(a)) is simplistic and also introduces a single point of failure (i.e., the coordinator). Extending the model to general hierarchies is probably not that difficult (even though effectively distributing the error bounds across the internal hierarchy nodes can be challenging [7]); however, extending the ideas to general, scalable distributed architectures (e.g., P2P networks) raises several theoretical and practical challenges. Second, while most of the proposed algorithmic tools have been prototyped and tested with real-life data streams, there is still a need for real system implementations that also address some of the key systems questions that arise (e.g., what functions and query language to support, how to interface to real users and applications, and so on). We have already started implementing some of the geometric monitoring ideas using Twitter’s Storm/ λ -architecture, and exploiting these ideas for large-scale, distributed Complex Event Processing (CEP) in the context of the FER-ARI project www.ferari-project.eu. Finally, from a more foundational perspective, there is a need for developing new models and theories for studying the complexity of such *continuous distributed computations*. These could build on the models of *communication complexity* [25] that study the complexity of distributed *one-shot* computations, perhaps combined with relevant ideas from information theory (e.g., distributed source coding). Some initial results in this direction have recently appeared for the case of simple norms and linear aggregates; see, e.g., [12, 22].

Acknowledgements This work was partially supported by the European Commission under ICT-FP7-FERARI (Flexible Event Processing for Big Data Architectures), www.ferari-project.eu.

References

1. N. Alon, P.B. Gibbons, Y. Matias, M. Szegedy, Tracking join and self-join sizes in limited storage, in *Proc. of the 18th ACM Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania (1999)
2. N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in *Proc. of the 28th Annual ACM Symposium on the Theory of Computing*, Philadelphia, Pennsylvania (1996), pp. 20–29
3. B. Babcock, C. Olston, Distributed top- k monitoring, in *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California (2003)
4. S. Burdakis, A. Deligiannakis, Detecting outliers in sensor networks using the geometric approach, in *Proc. of the 28th Intl. Conference on Data Engineering* (2012)
5. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in *Proc. of the Intl. Colloquium on Automata, Languages, and Programming*, Malaga, Spain (2002)
6. G. Cormode, M. Garofalakis, Streaming in a connected world: querying and tracking distributed data streams, in *2007 ACM SIGMOD Intl Conf. on Management of Data*, Beijing, China (2007). Tutorial
7. G. Cormode, M. Garofalakis, Approximate continuous querying over distributed streams. *ACM Trans. Database Syst.* **33**(2) (2008)
8. G. Cormode, M. Garofalakis, P.J. Haas, C. Jermaine, Synopses for massive data: samples, histograms, wavelets, sketches. *Found. Trends® Databases* **4**(1–3) (2012)
9. G. Cormode, M. Garofalakis, S. Muthukrishnan, R. Rastogi, Holistic aggregates in a networked world: distributed tracking of approximate quantiles, in *Proc. of the 2005 ACM SIGMOD Intl. Conference on Management of Data*, Baltimore, Maryland (2005)
10. G. Cormode, M. Garofalakis, D. Sacharidis, Fast approximate wavelet tracking on streams, in *Proc. of the 10th Intl. Conference on Extending Database Technology (EDBT'2006)*, Munich, Germany (2006)
11. G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, in *Proc. of the 22nd ACM Symposium on Principles of Database Systems*, San Diego, California (2003), pp. 296–306
12. G. Cormode, S. Muthukrishnan, K. Yi, Algorithms for distributed functional monitoring. *ACM Trans. Algorithms* **7**(2) (2011)
13. 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)
14. A. Das, S. Ganguly, M. Garofalakis, R. Rastogi, Distributed set-expression cardinality estimation, in *Proc. of the 30th Intl. Conference on Very Large Data Bases*, Toronto, Canada (2004)
15. A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi, Processing complex aggregate queries over data streams, in *Proc. of the 2002 ACM SIGMOD Intl. Conference on Management of Data*, Madison, Wisconsin (2002), pp. 61–72
16. S. Ganguly, M. Garofalakis, R. Rastogi, Processing set expressions over continuous update streams, in *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California (2003)
17. M. Garofalakis, D. Keren, V. Samoladas, Sketch-based geometric monitoring of distributed stream queries, in *Proc. of the 39th Intl. Conference on Very Large Data Bases*, Trento, Italy (2013)
18. N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, A. Schuster, Prediction-based geometric monitoring of distributed data streams, in *Proc. of the 2012 ACM SIGMOD Intl. Conference on Management of Data*, Scottsdale, Arizona (2012)
19. N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, A. Schuster, Distributed geometric query monitoring using prediction models. *ACM Trans. Database Syst.* **39**(2) (2014)
20. P.B. Gibbons, Distinct sampling for highly-accurate answers to distinct values queries and event reports, in *Proc. of the 27th Intl. Conference on Very Large Data Bases*, Rome, Italy (2001)

21. M.B. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in *Proc. of the 2001 ACM SIGMOD Intl. Conference on Management of Data*, Santa Barbara, California (2001)
22. Z. Huang, K. Yi, Q. Zhang, Randomized algorithms for tracking distributed count, frequencies, and ranks, in *Proc. of the 31st ACM Symposium on Principles of Database Systems* (2012)
23. R. Keralapura, G. Cormode, J. Ramamirtham, Communication-efficient distributed monitoring of thresholded counts, in *Proc. of the 2006 ACM SIGMOD Intl. Conference on Management of Data*, Chicago, Illinois (2006), pp. 289–300
24. D. Keren, I. Sharfman, A. Schuster, A. Livne, Shape-sensitive geometric monitoring. *IEEE Trans. Knowl. Data Eng.* **24**(8) (2012)
25. E. Kushilevitz, N. Nisan, *Communication Complexity* (Cambridge University Press, Cambridge, 1997)
26. A. Lazerson, I. Sharfman, D. Keren, A. Schuster, M. Garofalakis, V. Samoladas, Monitoring distributed streams using convex decompositions, in *Proc. of the 41st Intl. Conference on Very Large Data Bases* (2015)
27. S.R. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, The design of an acquisitional query processor for sensor networks, in *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California (2003)
28. G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in *Proc. of the 28th Intl. Conference on Very Large Data Bases*, Hong Kong, China (2002), pp. 346–357
29. S. Muthukrishnan, Data streams: algorithms and applications. *Found. Trends Theor. Comput. Sci.* **1**(2) (2005)
30. NII Shonan workshop on large-scale distributed computation, Shonan Village, Japan, January (2012). <http://www.nii.ac.jp/shonan/seminar011/>
31. C. Olston, J. Jiang, J. Widom, Adaptive filters for continuous queries over distributed data streams, in *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California (2003)
32. O. Papapetrou, M. Garofalakis, Continuous fragmented skylines over distributed streams, in *Proc. of the 30th Intl. Conference on Data Engineering*, Chicago, Illinois (2014)
33. I. Sharfman, A. Schuster, D. Keren, A geometric approach to monitoring threshold functions over distributed data streams, in *Proc. of the 2006 ACM SIGMOD Intl. Conference on Management of Data*, Chicago, Illinois (2006), pp. 301–312

Part IV

System Architectures and Languages

STREAM: The Stanford Data Stream Management System

Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom

1 Introduction

Traditional database management systems are best equipped to run *one-time* queries over finite stored data sets. However, many modern applications such as network monitoring, financial analysis, manufacturing, and sensor networks require long-running, or *continuous*, queries over continuous unbounded streams of data. In the *STREAM* project at Stanford, we are investigating data management and query processing for this class of applications. As part of the project we are building a general-purpose prototype *Data Stream Management System (DSMS)*, also called *STREAM*, that supports a large class of declarative continuous queries over continuous streams and traditional stored data sets. The *STREAM* prototype targets environments where streams may be rapid, stream characteristics and query loads may vary over time, and system resources may be limited.

Building a general-purpose DSMS poses many interesting challenges:

- Although we consider streams of structured data records together with conventional stored relations, we cannot directly apply standard relational semantics to complex continuous queries over this data. In Sect. 2, we describe the semantics and language we have developed for continuous queries over streams and relations.
- Declarative queries must be translated into *physical query plans* that are flexible enough to support optimizations and fine-grained scheduling decisions. Our query plans, composed of *operators*, *queues*, and *synopses*, are described in Sect. 3.

A. Arasu · B. Babcock · S. Babu · J. Cieslewicz · M. Datar · K. Ito · R. Motwani · U. Srivastava · J. Widom (✉)

Department of Computer Science, Stanford University, Stanford, CA, USA
e-mail: widom@cs.stanford.edu

- Achieving high performance requires that the DSMS exploit possibilities for sharing state and computation within and across query plans. In addition, constraints on stream data (e.g., ordering, clustering, referential integrity) can be inferred and used to reduce resource usage. In Sect. 4, we describe some of these techniques.
- Since data, system characteristics, and query load may fluctuate over the lifetime of a single continuous query, an *adaptive* approach to query execution is essential for good performance. Our continuous monitoring and reoptimization subsystem is described in Sect. 5.
- When incoming data rates exceed the DSMS’s ability to provide exact results for the active queries, the system should perform *load-shedding* by introducing approximations that gracefully degrade accuracy. Strategies for approximation are discussed in Sect. 6.
- Due to the long-running nature of continuous queries, DSMS administrators and users require tools to monitor and manipulate query plans as they run. This functionality is supported by our graphical interface described in Sect. 7.

Many additional problems, including exploiting parallelism and supporting crash recovery, are still under investigation. Future directions are discussed in Sect. 8.

2 The CQL Continuous Query Language

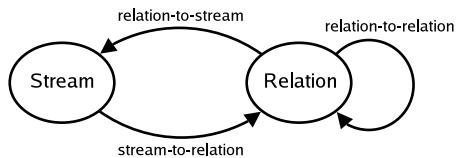
For simple continuous queries over streams, it can be sufficient to use a relational query language such as SQL, replacing references to relations with references to streams, and streaming new tuples in the result. However, as continuous queries grow more complex, e.g., with the addition of aggregation, subqueries, windowing constructs, and joins of streams and relations, the semantics of a conventional relational language applied to these queries quickly becomes unclear [3]. To address this problem, we have defined a formal *abstract semantics* for continuous queries, and we have designed *CQL*, a concrete declarative query language that implements the abstract semantics.

2.1 Abstract Semantics

The abstract semantics is based on two data types, *streams* and *relations*, which are defined using a discrete, ordered *time domain* Γ :

- A *stream* S is an unbounded bag (multiset) of pairs $\langle s, \tau \rangle$, where s is a tuple and $\tau \in \Gamma$ is the *timestamp* that denotes the logical arrival time of tuple s on stream S .
- A *relation* R is a time-varying bag of tuples. The bag of tuples at time $\tau \in \Gamma$ is denoted $R(\tau)$, and we call $R(\tau)$ an *instantaneous relation*. Note that our definition of a relation differs from the traditional one which has no built-in notion of time.

Fig. 1 Data types and operator classes in abstract semantics



The abstract semantics uses three classes of operators over streams and relations:

- A *relation-to-relation* operator takes one or more relations as input and produces a relation as output.
- A *stream-to-relation* operator takes a stream as input and produces a relation as output.
- A *relation-to-stream* operator takes a relation as input and produces a stream as output.

Stream-to-stream operators are absent—they are composed from operators of the above three classes. These three classes are “black box” components of our abstract semantics: the semantics does not depend on the exact operators in these classes, but only on generic properties of each class. Figure 1 summarizes our data types and operator classes.

A continuous query Q is a tree of operators belonging to the above classes. The inputs of Q are the streams and relations that are input to the leaf operators, and the output of Q is the output of the root operator. The output is either a stream or a relation, depending on the class of the root operator. At time τ , an operator of Q logically depends on its inputs up to τ : tuples of S_i with timestamps $\leq \tau$ for each input stream S_i , and instantaneous relations $R_j(\tau')$, $\tau' \leq \tau$, for each input relation R_j . The operator produces new outputs corresponding to τ : tuples of S with timestamp τ if the output is a stream S , or instantaneous relation $R(\tau)$ if the output is a relation R . The behavior of query Q is derived from the behavior of its operators in the usual inductive fashion.

2.2 Concrete Language

Our concrete declarative query language, *CQL* (for *Continuous Query Language*), is defined by instantiating the operators of our abstract semantics. Syntactically, CQL is a relatively minor extension to SQL.

Relation-to-Relation Operators in CQL

CQL uses SQL constructs to express its relation-to-relation operators, and much of the data manipulation in a typical CQL query is performed using these constructs, exploiting the rich expressive power of SQL.

Stream-to-Relation Operators in CQL

The stream-to-relation operators in CQL are based on the concept of a *sliding window* [5] over a stream, and are expressed using a window specification language derived from SQL-99:

- A *tuple-based sliding window* on a stream S takes an integer $N > 0$ as a parameter and produces a relation R . At time τ , $R(\tau)$ contains the N tuples of S with the largest timestamps $\leq \tau$. It is specified by following S with “[Rows N]”. As a special case, “[Rows Unbounded]” denotes the append-only window “[Rows ∞]”.
- A *time-based sliding window* on a stream S takes a time interval ω as a parameter and produces a relation R . At time τ , $R(\tau)$ contains all tuples of S with timestamps between $\tau - \omega$ and τ . It is specified by following S with “[Range ω]”. As a special case, “[Now]” denotes the window with $\omega = 0$.
- A *partitioned sliding window* on a stream S takes an integer N and a set of attributes $\{A_1, \dots, A_k\}$ of S as parameters, and is specified by following S with “[Partition By A_1, \dots, A_k Rows N]”. It logically partitions S into different substreams based on equality of attributes A_1, \dots, A_k , computes a tuple-based sliding window of size N independently on each substream, then takes the union of these windows to produce the output relation.

Relation-to-Stream Operators in CQL

CQL has three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream*. *Istream* (for “insert stream”) applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$, i.e., whenever s is inserted into R at time τ . *Dstream* (for “delete stream”) applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$, i.e., whenever s is deleted from R at time τ . *Rstream* (for “relation stream”) applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau)$, i.e., every current tuple in R is streamed at every time instant.

Example CQL Queries

Example 1 The following continuous query filters a stream S :

```
Select Istream(*) From S [Rows Unbounded] Where S.A > 10
```

Stream S is converted into a relation by applying an unbounded (append-only) window. The relation-to-relation filter “ $S.A > 10$ ” acts over this relation, and the inserts to the filtered relation are streamed as the result. CQL includes a number of syntactic shortcuts and defaults for convenience, which permit the above query to be rewritten in the following more intuitive form:

```
Select * From S Where S.A > 10
```

Example 2 The following continuous query is a *windowed join* of two streams S_1 and S_2 :

```
Select * From S1 [Rows 1000], S2 [Range 2 Minutes]
Where S1.A = S2.A And S1.A > 10
```

The answer to this query is a relation. At any given time, the answer relation contains the join (on attribute A with $A > 10$) of the last 1000 tuples of S_1 with the tuples of S_2 that have arrived in previous 2 minutes. If we prefer instead to produce a stream containing new A values as they appear in the join, we can write “`Istream(S1.A)`” instead of “`*`” in the `Select` clause.

Example 3 The following continuous query probes a stored table R based on each tuple in stream S and streams the result:

```
Select Rstream(S.A, R.B) From S [Now], R Where S.A = R.A
```

Complete details of CQL including syntax, semantic foundations, syntactic shortcuts and defaults, equivalences, and a comparison against related continuous query languages are given in [3].

3 Query Plans and Execution

When a continuous query specified in CQL is registered with the STREAM system, a *query plan* is compiled from it. Query plans are composed of *operators*, which perform the actual processing, *queues*, which buffer tuples (or references to tuples) as they move between operators, and *synopses*, which store operator state.

3.1 Operators

Recall from Sect. 2 that there are two fundamental data types in our query language: streams, defined as bags of tuple-timestamp pairs, and relations, defined as time-varying bags of tuples. We unify these two types in our implementation as sequences of timestamped tuples, where each tuple additionally is flagged as either an *insertion* (+) or *deletion* (-). We refer to the tuple-timestamp-flag triples as *elements*.

Streams only include + elements, while relations may include both + and - elements to capture the changing relation state over time. Queues logically contain sequences of elements representing either streams or relations. Each query plan operator reads from one or more *input queues*, processes the input based on its semantics, and writes any output to an *output queue*. Individual operators may materialize their relational inputs in synopses (see Sect. 3.3) if such state is useful.

The operators in the STREAM system that implement the CQL language are summarized in Table 1. In addition, there are several *system operators* to handle

Table 1 Operators used in STREAM query plans

Name	Operator type	Description
select	relation-to-relation	Filters elements based on predicate(s)
project	relation-to-relation	Duplicate-preserving projection
binary-join	relation-to-relation	Joins two input relations
mjoin	relation-to-relation	Multiway join from [22]
union	relation-to-relation	Bag union
except	relation-to-relation	Bag difference
intersect	relation-to-relation	Bag intersection
antisemijoin	relation-to-relation	Antisemijoin of two input relations
aggregate	relation-to-relation	Performs grouping and aggregation
duplicate-eliminate	relation-to-relation	Performs duplicate elimination
seq-window	stream-to-relation	Implements time-based, tuple-based, and partitioned windows
i-stream	relation-to-stream	Implements <i>Istream</i> semantics
d-stream	relation-to-stream	Implements <i>Dstream</i> semantics
r-stream	relation-to-stream	Implements <i>Rstream</i> semantics

“housekeeping” tasks such as marshaling input and output and connecting query plans together. During execution, operators are *scheduled* individually, allowing for fine-grained control over queue sizes and query latencies. Scheduling algorithms are discussed later in Sect. 4.3.

3.2 Queues

A queue in a query plan connects its “producing” plan operator O_P to its “consuming” operator O_C . At any time a queue contains a (possibly empty) collection of elements representing a portion of a stream or relation. The elements that O_P produces are inserted into the queue and buffered there until they are processed by O_C .

Many of the operators in our system require that elements on their input queues be read in nondecreasing timestamp order. Consider, for example, a window operator O_W on a stream S as described in Sect. 2.2. If O_W receives an element $\langle s, \tau, + \rangle$ and its input queue is guaranteed to be in nondecreasing timestamp order, then O_W knows it has received all elements with timestamp $\tau' < \tau$, and it can construct the state of the window at time $\tau - 1$. (If timestamps are known to be unique it can construct the state at time τ .) If, on the other hand, O_W does not have this guarantee, it can never be sure it has enough information to construct any window correctly. Thus, we require all queues to enforce nondecreasing timestamps.

Mechanisms for buffering tuples and generating *heartbeats* to ensure nondecreasing timestamps, without sacrificing correctness or completeness, are discussed in detail in [17].

3.3 Synopses

Logically, a *synopsis* belongs to a specific plan operator, storing state that may be required for future evaluation of that operator. (In our implementation, synopses are shared among operators whenever possible, as described later in Sect. 4.1.) For example, to perform a windowed join of two streams, the join operator must be able to probe all tuples in the current window on each input stream. Thus, the join operator maintains one synopsis (e.g., a hash table) for each of its inputs. On the other hand, operators such as selection and duplicate-preserving union do not require any synopses.

The most common use of a synopsis in our system is to materialize the current state of a (derived) relation, such as the contents of a sliding window or the relation produced by a subquery. Synopses also may be used to store a summary of the tuples in a stream or relation for approximate query answering, as discussed later in Sect. 6.2.

Performance requirements often dictate that synopses (and queues) must be kept in memory, and we tacitly make that assumption throughout this chapter. Our system does support overflow of these structures to disk, although currently it does not implement sophisticated algorithms for minimizing I/O when overflow occurs; see, e.g., [20].

3.4 Example Query Plan

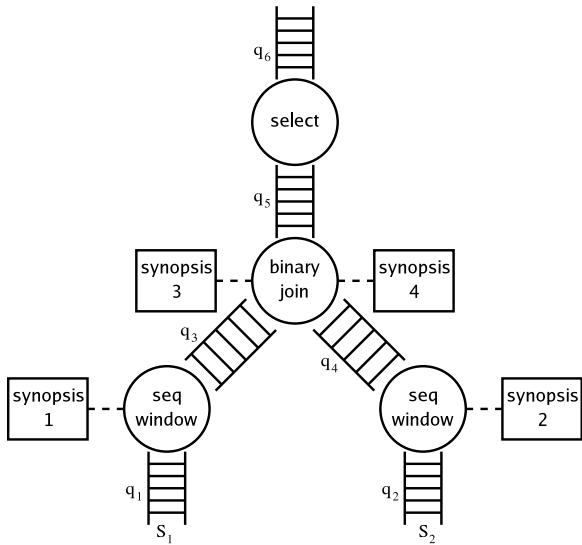
When a CQL query is registered, STREAM constructs a query plan: a tree of operators, connected by queues, with synopses attached to operators as needed. As a simple example, a plan for the query from Example 2 is shown in Fig. 2. The original query is repeated here for convenience:

```
Select * From S1 [Rows 1000], S2 [Range 2 Minutes]
Where S1.A = S2.A And S1.A > 10
```

There are four operators in the example plan: a `select`, a `binary-join`, and one instance of `seq-window` for each input stream. Queues q_1 and q_2 hold the input stream elements which could, for example, have been received over the network and placed into queues by a system operator (not depicted). Queue q_3 , which is the output queue of the (stream-to-relation) operator `seq-window`, holds elements representing the relation “S1 [Rows 1000].” Queue q_4 holds elements for “S2 [Range 2 Minutes].” Queue q_5 holds elements for the joined relation “ $S1 \text{ [Rows 1000]} \bowtie S2 \text{ [Range 2 Minutes]}$,” and from these elements, Queue q_6 holds the elements passing the `select` operator. q_6 may lead to an output operator sending elements to the application, or to another query plan operator within the system.

The `select` operator can be pushed down into one or both branches below the `binary-join` operator, and also below the `seq-window` operator on $S2$.

Fig. 2 A simple query plan illustrating operators, queues, and synopses



However, tuple-based windows do not commute with filter conditions, and therefore the `select` operator cannot be pushed below the `seq-window` operator on S_1 .

The plan has four synopses, $synopsis_1$ – $synopsis_4$. Each `seq-window` operator maintains a synopsis so that it can generate “–” elements when tuples expire from the sliding window. The `binary-join` operator maintains a synopsis materializing each of its relational inputs for use in performing joins with tuples on the opposite input, as described earlier. Since the `select` operator does not need to maintain any state, it does not have a synopsis.

Note that the contents of $synopsis_1$ and $synopsis_3$ are similar (as are the contents of $synopsis_2$ and $synopsis_4$), since both maintain a materialization of the same window, but at slightly different positions of stream S_1 . Section 4.1 discusses how we eliminate such redundancy.

3.5 Query Plan Execution

When a query plan is executed, a *scheduler* selects operators in the plan to execute in turn. The semantics of each operator depends only on the timestamps of the elements it processes, not on system or “wall-clock” time. Thus, the order of execution has no effect on the data in the query result, although it can affect other properties such as latency and resource utilization. Scheduling is discussed further in Sect. 4.3.

Continuing with our example from the previous section, the `seq-window` operator on S_1 , on being scheduled, reads stream elements from q_1 . Suppose it reads element $\langle s, \tau, + \rangle$. It inserts tuple s into $synopsis_1$, and if the window is full (i.e., the synopsis already contains 1000 tuples), it removes the earliest tuple s' in the

synopsis. It then writes output elements into q_3 : the element $\langle s, \tau, + \rangle$ to reflect the addition of s to the window, and the element $\langle s', \tau, - \rangle$ to reflect the deletion of s' as it exits the window. Both of these events occur logically at the same time instant τ . The other seq-window operator is analogous.

When scheduled, the binary-join operator reads the earliest element across its two input queues. If it reads an element $\langle s, \tau, + \rangle$ from q_3 , then it inserts s into $synopsis_3$ and joins s with the contents of $synopsis_4$, generating output elements $\langle s \cdot t, \tau, + \rangle$ for each matching tuple t in $synopsis_4$. Similarly, if the binary-join operator reads an element $\langle s, \tau, - \rangle$ from q_3 , it generates $\langle s \cdot t, \tau, - \rangle$ for each matching tuple t in $synopsis_4$. A symmetric process occurs for elements read from q_4 . In order to ensure that the timestamps of its output elements are nondecreasing, the binary-join operator must process its input elements in nondecreasing timestamp order across both inputs.

Since the select operator is stateless, it simply dequeues elements from q_5 , tests the tuple against its selection predicate, and enqueues the identical element into q_6 if the test passes, discarding it otherwise.

4 Performance Issues

In the previous section, we introduced the basic architecture of our query processing engine. However, simply generating the straightforward query plans and executing them as described can be very inefficient. In this section, we discuss ways in which we improve the performance of our system by eliminating data redundancy (Sect. 4.1), selectively discarding data that will not be used (Sect. 4.2), and scheduling operators to most efficiently reduce intermediate state (Sect. 4.3).

4.1 Synopsis Sharing

In Sect. 3.4, we observed that multiple synopses within a single query plan may materialize nearly identical relations. In Fig. 2, $synopsis_1$ and $synopsis_3$ are an example of such a pair.

We eliminate this redundancy by replacing the two synopses with lightweight *stubs*, and a single *store* to hold the actual tuples. These stubs implement the same interfaces as non-shared synopses, so operators can be oblivious to the details of sharing. As a result, synopsis sharing can be enabled or disabled on the fly.

Since operators are scheduled independently, it is likely that operators sharing a single synopsis store will require slightly different views of the data. For example, if queue q_3 in Fig. 2 contains 10 elements, then $synopsis_3$ will not reflect these changes (since the binary-join operator has not yet processed them), although $synopsis_1$ will. When synopses are shared, logic in the store tracks the progress of each stub, and presents the appropriate view (subset of tuples) to each of the stubs. Clearly, the

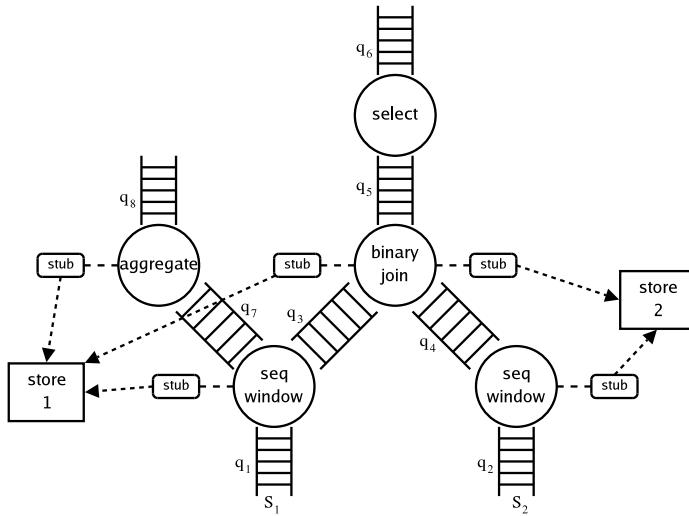


Fig. 3 A query plan illustrating synopsis sharing

store must contain the union of its corresponding stubs: A tuple is inserted into the store as soon as it is inserted by any one of the stubs, and it is removed only when it has been removed from all of the stubs.

To further decrease state redundancy, multiple query plans involving similar intermediate relations can share synopses as well. For example, suppose the following query is registered in addition to the query in Sect. 3.4:

```
Select A, Max(B) From S1 [Rows 200] Group By A
```

Since sliding windows are contiguous in our system, the window on $S1$ in this query is a subset of the window on $S1$ in the other query. Thus, the same data store can be used to materialize both windows. The combination of the two query plans with both types of sharing is illustrated in Fig. 3.

4.2 Exploiting Constraints

Streams may exhibit certain data or arrival patterns that can be exploited to reduce run-time synopsis sizes. Such *constraints* can either be specified explicitly at stream-registration time, or inferred by gathering statistics over time [6]. (An alternate and more dynamic technique is for the streams to contain *punctuations*, which specify run-time constraints that also can be used to reduce resource requirements [21].)

As a simple example, consider a continuous query that joins a stream *Orders* with a stream *Fulfillments* based on attributes *orderID* and *itemID*, perhaps to monitor average fulfillment delays. In the general case, answering this query precisely requires synopses of unbounded size [2]. However, if we know that all elements for

a given $orderID$ and $itemID$ arrive on $Orders$ before the corresponding elements arrive on $Fulfillments$, then we need not maintain a join synopsis for the $Fulfillments$ operand at all. Furthermore, if $Fulfillments$ elements arrive clustered by $orderID$, then we need only save $Orders$ tuples for a given $orderID$ until the next $orderID$ is seen.

We have identified several types of useful constraints over data streams. Effective optimizations can be made even when the constraints are not strictly met by defining an *adherence parameter*, k , that captures how closely a given stream or pair of streams adheres to a constraint of that type. We refer to these as k -constraints:

- A *referential integrity k-constraint* on a many-one join between streams defines a bound k on the delay between the arrival of a tuple on the “many” stream and the arrival of its joining “one” tuple on the other stream.
- An *ordered-arrival k-constraint* on a stream attribute $S.A$ defines a bound k on the amount of reordering in values of $S.A$. Specifically, given any tuple s in stream S , for all tuples s' that arrive at least $k + 1$ elements after s , it must be true that $s'.A \geq s.A$.
- A *clustered-arrival k-constraint* on a stream attribute $S.A$ defines a bound k on the distance between any two elements that have the same value of $S.A$.

We have developed query plan construction and execution algorithms that take stream constraints into account in order to reduce synopsis sizes at query operators by discarding unnecessary state [9]. The smaller the value of k for each constraint, the more state that can be discarded. Furthermore, if an assumed k -constraint is not satisfied by the data, our algorithm produces an approximate answer whose error is proportional to the degree of deviation of the data from the constraint.

4.3 Operator Scheduling

An operator consumes elements from its input queues and produces elements on its output queue. Thus, the global operator scheduling policy can have a large effect on memory utilization, particularly with bursty input streams.

Consider the following simple example. Suppose we have a query plan with two operators, O_1 followed by O_2 . Assume that O_1 takes one time unit to process a batch of n elements, and it produces $0.2n$ output elements per input batch (i.e., its *selectivity* is 0.2). Further, assume that O_2 takes one time unit to operate on $0.2n$ elements, and it sends its output out of the system. (As far as the system is concerned, O_2 produces no elements, and therefore its selectivity is 0.) Consider the following bursty arrival pattern: n elements arrive at every time instant from $t = 0$ to $t = 6$, then no elements arrive from time $t = 7$ through $t = 13$.

Under this scenario, consider the following scheduling strategies:

- *FIFO scheduling*. When batches of n elements have been accumulated, they are passed through both operators in two consecutive time units, during which no other element is processed.

- *Greedy scheduling.* At any time instant, if there is a batch of n elements buffered before O_1 , it is processed in one time unit. Otherwise, if there are more than $0.2n$ elements buffered before O_2 , then $0.2n$ elements are processed using one time unit. This strategy is “greedy” since it gives preference to the operator that has the greatest rate of reduction in total queue size per unit time.

The following table shows the expected total queue size for each strategy, where each table entry is a multiplier for n :

Time	0	1	2	3	4	5	6	Avg
FIFO scheduling	1.0	1.2	2.0	2.2	3.0	3.2	4.0	2.4
Greedy scheduling	1.0	1.2	1.4	1.6	1.8	2.0	2.2	1.6

After time $t = 6$, input queue sizes for both strategies decline until they reach 0 after time $t = 13$. The greedy strategy performs better because it runs O_1 whenever it has input, reducing queue size by $0.8n$ elements each time step, while the FIFO strategy alternates between executing O_1 and O_2 .

However, the greedy algorithm has its shortcomings. Consider a plan with operators O_1 , O_2 , and O_3 . O_1 produces $0.9n$ elements per n input elements in one time unit, O_2 processes $0.9n$ elements in one time unit without changing the input size (i.e., it has selectivity 1), and O_3 processes $0.9n$ elements in one time unit and sends its output out of the system (i.e., it has selectivity 0). Clearly, the greedy algorithm will prioritize O_3 first, followed by O_1 , and then O_2 . If we consider the arrival pattern in the previous example then our total queue size is as follows (again as multipliers for n):

Time	0	1	2	3	4	5	6	Avg
FIFO scheduling	1.0	1.9	2.9	3.0	3.9	4.9	5.0	3.2
Greedy scheduling	1.0	1.9	2.8	3.7	4.6	5.5	6.4	3.7

In this case, the FIFO algorithm is better. Under the greedy strategy, although O_3 has highest priority, sometimes it is “blocked” from running because it is preceded by O_2 , the operator with the lowest priority. If O_1 , O_2 and O_3 are viewed as a single block, then together they reduce n elements to zero elements over three units of time, for an average reduction of $0.33n$ elements per unit time—better than the reduction rate of $0.1n$ elements O_1 provides. Since the greedy algorithm considers individual operators only, it does not take advantage of this fact.

This observation forms the basis of our *chain scheduling* algorithm [4]. Our algorithm forms blocks (“chains”) of operators as follows: Start by marking the first operator in the plan as the “current” operator. Next, find the block of consecutive operators starting at the “current” operator that maximizes the reduction in total queue size per unit time. Mark the first operator following this block as the “current” operator and repeat the previous step until all operators have been assigned to

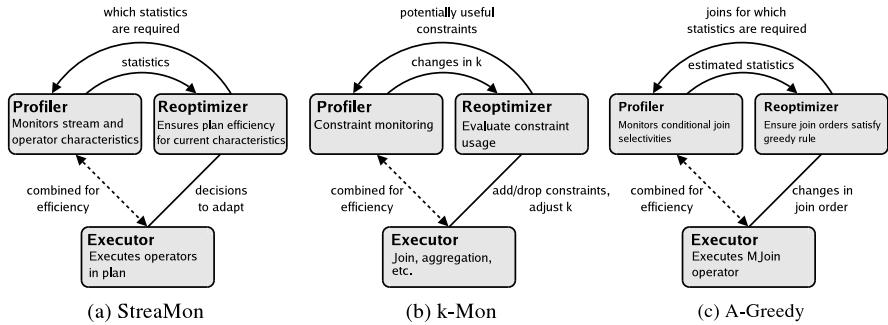


Fig. 4 Adaptive query processing

chains. Chains are scheduled according to the greedy algorithm, but within a chain, execution proceeds in FIFO order. In terms of overall memory usage, this strategy is provably close to the optimal “clairvoyant” scheduling strategy, i.e., the optimal strategy based on knowledge of future input [4].

5 Adaptivity

In long-running stream applications, data and arrival characteristics of streams may vary significantly over time [13]. Query loads and system conditions may change as well. Without an *adaptive* approach to query processing, performance may drop drastically over time as the environment changes. The STREAM system includes a monitoring and adaptive query processing infrastructure called *StreaMon* [10].

StreaMon has three components as shown in Fig. 4(a): an *Executor*, which runs query plans to produce results, a *Profiler*, which collects and maintains statistics about stream and plan characteristics, and a *Reoptimizer*, which ensures that the plans and memory structures are the most efficient for current characteristics. In many cases, we combine the profiler and executor to reduce the monitoring overhead.

The Profiler and Reoptimizer are essential for adaptivity, but they compete for resources with the Executor. We have identified a clear three-way tradeoff among run-time overhead, speed of adaptivity, and provable convergence to good strategies if conditions stabilize. StreaMon supports multiple adaptive algorithms that lie at different points along this tradeoff spectrum.

StreaMon can detect useful k -constraints (recall Sect. 4.2) in streams and exploit them to reduce memory requirements for many continuous queries. In addition, it can adaptively adjust the adherence parameter k based on the actual data in the streams. Figure 4(b) shows the portions of StreaMon’s Profiler and Reoptimizer that handle k -constraints, referred to as *k-Mon*. When a query is registered, the optimizer notifies the profiler of potentially useful constraints. As the executor runs the query, the profiler monitors the input streams continuously and informs the reoptimizer whenever it detects a change in a k value for any of these constraints. The

reoptimizer component adapts to these changes by adding or dropping constraints used by the executor and adjusting k values used for memory allocation.

StreaMon also implements an algorithm called *Adaptive Greedy* (or *A-Greedy*) [7], which maintains join orders adaptively for pipelined multiway stream joins, also known as *MJoins* [22]. Figure 4(c) shows the portions of StreaMon’s Profiler and Reoptimizer that comprise the A-Greedy algorithm. Using A-Greedy, StreaMon monitors conditional selectivities and orders stream joins to minimize overall work in current conditions. In addition, StreaMon detects when changes in conditions may have rendered current orderings suboptimal, and reorders in those cases. In stable conditions, the orderings converged on by the A-Greedy algorithm are equivalent to those selected by a static Greedy algorithm that is provably within a cost factor < 4 of optimal. In practice, the Greedy algorithm, and therefore A-Greedy, nearly always finds the optimal orderings.

In addition to adaptive join ordering, we use StreaMon to adaptively add and remove *subresult caches* in stream join plans, to avoid recomputation of intermediate results [8]. StreaMon monitors costs and benefits of candidate caches, selects caches to use, allocates memory to caches, and adapts over the entire spectrum between stateless MJoins and cache-rich join trees, as stream and system conditions change.

Currently we are in the process of applying the StreaMon approach to make even more aspects of the STREAM system adaptive, including sharing of synopses and subplans, and operator scheduling.

6 Approximation

In many applications data streams can be bursty, with unpredictable peaks during which the load may exceed available system resources, especially if numerous complex queries have been registered. Fortunately, for many stream applications (e.g., in many monitoring tasks), it is acceptable to degrade accuracy gracefully by providing approximate answers during load spikes [18].

There are two primary ways in which a DSMS may be resource-limited:

- *CPU-limited* (Sect. 6.1)—The data arrival rate may be so high that there is insufficient CPU time to process each stream element. In this case, the system may approximate by dropping elements before they are processed.
- *Memory-limited* (Sect. 6.2)—The total state required for all registered queries may exceed available memory. In this case, the system may selectively retain some state, discarding the rest.

6.1 CPU-Limited Approximation

CPU usage can be reduced by *load-shedding*—dropping elements from query plans and saving the CPU time that would be required to process them to completion. We

implement load-shedding by introducing *sampling* operators that probabilistically drop stream elements as they are input to the query plan.

The time-accuracy tradeoffs for sampling are more understandable for some query plans than others. For example, if we know a few basic statistics on the distribution of values in our streams, probabilistic guarantees on the accuracy of sliding-window aggregation queries for a given sampling rate can be derived mathematically, as we will show in below. However, in more complex queries—ones involving joins, for example—the error introduced by sampling is less clear and the choice of error metric may be application-dependent.

Suppose we have a set of sliding-window aggregation queries over the input streams. A simple example is

```
Select Avg (Temp) From SensorReadings [Range 5 Minutes]
```

If we have many such queries in a CPU-limited setting, our goal is to sample the inputs so as to minimize the maximum relative error across all queries. (As an extension, we can weight the relative errors to provide “quality-of-service” distinctions.) It follows that we should select sampling rates such that the relative error is the same for all queries. Assume that for a given query Q_i we know the mean μ_i and standard deviation σ_i of the values we are aggregating, as well as the window size N_i . These statistics can be collected by the profiler component in the *StreaMon* architecture (recall Sect. 5). We can use the *Hoeffding inequality* [16] to derive a bound on the probability δ that our relative error exceeds a given threshold ϵ_{max} for a given sampling rate. We then fix δ at a low value (e.g., 0.01) and algebraically manipulate this equation to derive the required sampling rate P_i [6],

$$P_i = \frac{1}{\epsilon_{max}} \sqrt{\frac{\sigma_i^2 + \mu_i^2}{2N_i\mu_i^2} \log \frac{2}{\delta}}.$$

Our load-shedding policy solves for the best achievable ϵ_{max} given the constraint that the system, after inserting load-shedders, can keep up with the arrival of elements. It then adds sampling operators at various points in the query plan such that effective sampling rate for a query Q_i is P_i .

6.2 Memory-Limited Approximation

Even using our scheduling algorithm that minimizes memory devoted to queues (Sect. 4.3), and our constraint-aware execution strategy that minimizes synopsis sizes (Sect. 4.2), if we have many complex queries with large windows (e.g., large tuple-based windows, or any size time-based windows over rapid data streams), memory may become a constraint. Spilling to disk may not be a feasible option due to online performance requirements.

In this scenario, memory usage can be reduced at the cost of accuracy by reducing the size of synopses at one or more operators. Incorporating a window into a synopsis where no window is being used, or shrinking the existing window, will

shrink the synopsis. Note that if sharing is in place (Sect. 4.1), then modifying a single synopsis may affect multiple queries.

Reducing the size of a synopsis generally tends to also reduce the sizes of synopses above it in the query plan, but there are exceptions. Consider a query plan where a sliding-window synopsis is used by a duplicate-elimination operator. Shrinking the window size can increase the operator’s output rate, leading to an increase in the size of “later” synopses. Fortunately, most of these cases can be detected statically when the query plan is generated, and the system can avoid reducing synopsis sizes in such cases.

There are other methods for reducing synopsis size, including maintaining a sample of the intended synopsis content (which is not always equivalent to inserting a sample operator into the query plan), using histograms [19] or wavelets [12] when the synopsis is used for aggregation or even for a join, and using Bloom filters [11] for duplicate elimination, set difference, or set intersection. In addition, synopsis sizes can be reduced by lowering the k values for known k -constraints (Sect. 4.2). Lower k values cause more state to be discarded, but result in loss of accuracy if the constraint does not hold for the assumed k . All of these techniques share the property that memory use is flexible, and it can be traded against precision statically or dynamically.

See Sect. 8.3 for discussion on future directions related to approximation.

7 The STREAM System Interface

In a system for continuous queries, it is important for users, system administrators, and system developers to have the ability to inspect the system while it is running and to experiment with adjustments. To meet these needs, we have developed a graphical *query and system visualizer* for the STREAM system. The visualizer allows the user to:

- View the structure of query plans and their component *entities* (operators, queues, and synopses). Users can view the path of data flow through each query plan as well as the sharing of computation and state within the plan.
- View the detailed properties of each entity. For example, the user can inspect the amount of memory being used (for queue and synopsis entities), the current throughput (for queue and operator entities), selectivity of predicates (for operator entities), and other properties.
- Dynamically adjust entity properties. These changes are reflected in the system in real time. For example, an administrator may choose to increase the size of a queue to better handle bursty arrival patterns.
- View *monitoring graphs* that display time-varying entity properties such as queue sizes, throughput, overall memory usage, and join selectivity, plotted dynamically against time.

A screenshot of our visualizer is shown in Fig. 5. The large pane at the left displays a graphical representation of a currently selected query plan. The particular

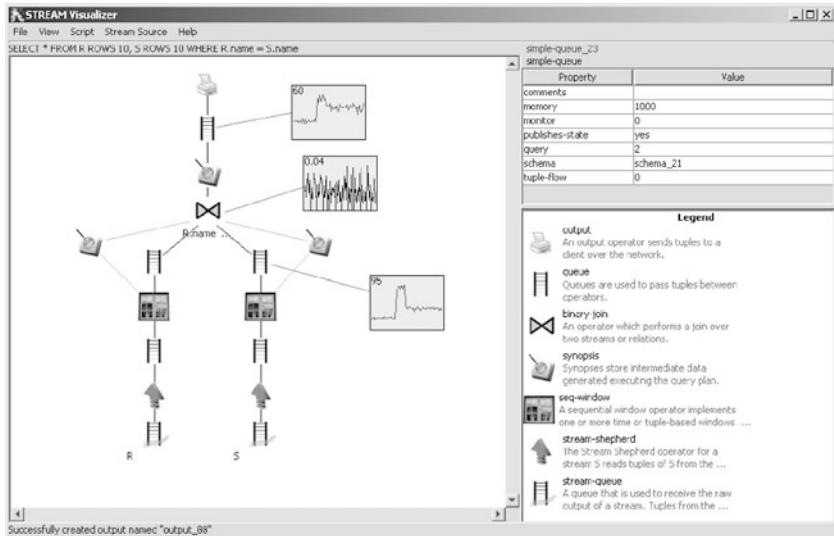


Fig. 5 Screenshot of the STREAM visualizer

query shown is a windowed join over two streams, R and S . Each entity in the plan is represented by an icon: the ladder-shaped icons are queues, the boxes with magnifying glasses over them are synopses, the window panes are windowing operators, and so on. In this example, the user has added three monitoring graphs: the rate of element flow through queues above and below the join operator, and the selectivity of the join.

The upper-right pane displays the property-value table for a currently selected entity. The user can inspect this list and can alter the values of some of the properties interactively. Finally, the lower-right pane displays a legend of entity icons and descriptions for reference.

Our technique for implementing the monitoring graphs shown in Fig. 5 is based on *introspection queries* on a special system stream called *SysStream*. Every entity can publish any of its property values at any time onto *SysStream*. When a specific dynamic monitoring task is desired, e.g., monitoring recent join selectivity, the relevant entity writes its statistics periodically on *SysStream*. Then a standard CQL query, typically a windowed aggregation query, is registered over *SysStream* to compute the desired continuous result, which is fed to the monitoring graph in the visualizer. Users and applications can also register arbitrary CQL queries over *SysStream* for customized monitoring tasks.

8 Future Directions

At the time of writing we plan to pursue the following general directions of future work.

8.1 Distributed Stream Processing

So far we have considered a *centralized* DSMS model where all processing takes place at a single system. In many applications, the stream data is actually produced at distributed *sources*. Moving some processing to the sources instead of moving all data to a central system may lead to more efficient use of processing and network resources. Many new challenges arise if we wish to build a fully distributed data stream system with capabilities equivalent to our centralized system.

8.2 Crash Recovery

The ability to recover to a consistent state following a system crash is a key feature of conventional database systems, but has yet to be investigated for data stream systems. There are some fundamental differences between DBMSs and DSMSs that play an important role in crash recovery:

- The notion of consistent state in a DBMS is defined based on *transactions*, which are closely tied to the conventional one-time query model. ACID transactional properties do not map directly to the continuous query paradigm.
- In a DBMS, the data in the database cannot change during down-time. In contrast, many stream applications deliver data to the DSMS from outside sources that do not stop generating data while the system is down, possibly requiring the DSMS to “catch up” following a crash.
- In a DBMS, queries underway at the time of a crash may be forgotten—it is the responsibility of the application to restart them. In contrast, registered continuous queries are part of the persistent state of a DSMS.

These differences lead us to believe that new mechanisms are needed for crash recovery in data stream systems. While logging of some type and perhaps even some notion of transactions may form a component of the solution, new techniques will be required as well.

8.3 Improved Approximation

Although some aspects of the approximation problem have already been addressed (see Sect. 6), more work is needed to address the problem in its full generality. In the memory-limited case, work is needed on the problem of sampling over arbitrary sub-queries, computing “maximum-subset” as opposed to sampling approximations, and maximizing accuracy over multiple weighted queries. In the CPU-limited case, we need to address a broader range of queries, especially considering joins. Finally, we need to handle situations when the DSMS may be both CPU and memory-limited.

A significant challenge related to approximation is developing mechanisms whereby the system can indicate to users or applications that approximation is occurring, and to what degree. The converse is also important: mechanisms for users to indicate acceptable degrees of approximation. As one step in the latter direction, we are developing extensions to CQL that enable the specification of “approximation guidelines” so that the user can indicate acceptable tolerances and priorities.

8.4 Relationship to Publish–Subscribe Systems

In a *publish–subscribe (pub–sub)* system (see, e.g., [1, 14, 15]), events may be published continuously, and they are forwarded by the system to users who have registered matching subscriptions. Clearly, we can map a pub–sub system to a DSMS by considering publications as streams and subscriptions as continuous queries. However, the techniques we have developed so far for processing continuous queries in a DSMS have been geared primarily toward a relatively small number of independent, complex queries, while a pub–sub system has potentially millions of simple, similar queries. We are exploring techniques to bridge the capabilities of the two: From the pub–sub perspective, provide a system that supports a more general model of subscriptions. From the DSMS perspective, extend our approach to scale to an extremely larger number of queries.

References

1. M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, T.D. Chandra, Matching events in a content-based subscription system, in *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing* (1999), pp. 53–61
2. A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom, Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Syst.* **29**(1), 1–33 (2004)
3. A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution. *VLDB J.* **15**(2), 121–142 (2006)
4. B. Babcock, S. Babu, M. Datar, 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* (2003), pp. 253–264
5. B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in *Proc. of the 21st ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems* (2002), pp. 1–16
6. B. Babcock, M. Datar, R. Motwani, Load shedding for aggregation queries over data streams, in *Proc. of the 20th Intl. Conf. on Data Engineering* (2004)
7. S. Babu, R. Motwani, K. Munagala, I. Nishizawa, J. Widom, Adaptive ordering of pipelined stream filters, in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data* (2004)
8. S. Babu, K. Munagala, J. Widom, R. Motwani, Adaptive caching for continuous queries, in *Proc. of the 21st Intl. Conf. on Data Engineering* (2005), pp. 118–129
9. S. Babu, U. Srivastava, J. Widom, Exploiting k -constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.* **29**(3), 545–580 (2004)
10. S. Babu, J. Widom, StreaMon: an adaptive engine for stream query processing, in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data* (2004). Demonstration description

11. B.H. Bloom, Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **13**(7), 422–426 (1970)
12. K. Chakrabarti, M.N. Garofalakis, R. Rastogi, K. Shim, Approximate query processing using wavelets, in *Proc. of the 26th Intl. Conf. on Very Large Data Bases* (2000), pp. 111–122
13. J. Gehrke (ed.), Data stream processing. *IEEE Comput. Soc. Bull. Technical Comm. Database Eng.* **26**(1) (2003)
14. F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, D. Shasha, Filtering algorithms and implementation for very fast publish/subscribe, in *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data* (2001), pp. 115–126
15. R.E. Gruber, B. Krishnamurthy, E. Panagos, READY: a high performance event notification system, in *Proc. of the 16th Intl. Conf. on Data Engineering* (2000), pp. 668–669
16. W. Hoeffding, Probability inequalities for sums of bounded random variables. *J. Am. Stat. Soc.* **58**(301), 13–30 (1963)
17. U. Srivastava, J. Widom, Flexible time management in data stream systems, in *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (2004)
18. N. Tatbul, U. Cetintemel, S.B. Zdonik, M. Cherniak, M. Stonebraker, Load shedding in a data stream manager, in *Proc. of the 29th Intl. Conf. on Very Large Data Bases* (2003), pp. 309–320
19. N. Thaper, S. Guha, P. Indyk, N. Koudas, Dynamic multidimensional histograms, in *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data* (2002), pp. 428–439
20. D. Thomas, R. Motwani, Caching queues in memory buffers, in *Proc. of the 15th Annual ACM-SIAM Symp. on Discrete Algorithms* (2004)
21. P.A. Tucker, D. Maier, T. Sheard, L. Fegaras, Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.* **15**(3), 555–568 (2003)
22. S. Viglas, J.F. Naughton, J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, in *Proc. of the 29th Intl. Conf. on Very Large Data Bases* (2003), pp. 285–296

The Aurora and Borealis Stream Processing Engines

Uğur Çetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan,
Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang,
Samuel Madden, Anurag Maskey, Alexander Rasin, Esther Ryvkina,
Mike Stonebraker, Nesime Tatbul, Ying Xing, and Stan Zdonik

1 Introduction and History

Over the last several years, a great deal of progress has been made in the area of stream-processing engines (SPEs) [9, 11, 17]. Three basic tenets distinguish SPEs from current data processing engines. First, they must support primitives for streaming applications. Unlike Online Transaction Processing (OLTP), which processes messages in isolation, streaming applications entail time series operations on streams of messages. Although a time series “blade” was added to the Illustra Object-Relational DBMS, generally speaking, time series operations are not well supported by current DBMSs. Second, streaming applications entail a real-time component. If one is content to see an answer later, then one can store incoming messages in a data warehouse and run a historical query on the warehouse to find information of interest. This tactic does not work if the answer must be constructed in real time. The need for real-time answers also dictates a fundamentally different storage architecture. DBMSs universally store and index data records before making them available for query activity. Such outbound processing, where data are stored before being processed, cannot deliver real-time latency, as required by SPEs. To meet more stringent latency requirements, SPEs must adopt an alternate model, which we refer to as “inbound processing”, where query processing is performed

U. Çetintemel (✉) · Y. Ahmad · J.-H. Hwang · A. Rasin · N. Tatbul · Y. Xing · S. Zdonik
Department of Computer Science, Brown University, Providence, RI, USA
e-mail: ugur@cs.brown.edu

D. Abadi · H. Balakrishnan · M. Balazinska · S. Madden · M. Stonebraker
Department of EECS and Laboratory for Computer Science,
Massachusetts Institute of Technology, Cambridge, MA, USA

M. Cherniack · A. Maskey · E. Ryvkina
Department of Computer Science, Brandeis University, Waltham, MA, USA

directly on incoming messages before (or instead of) storing them. Lastly, an SPE must have capabilities to gracefully deal with spikes in message load. Incoming traffic is usually bursty, and it is desirable to selectively degrade the performance of the applications running on an SPE.

The Aurora stream-processing engine, motivated by these three tenets, is currently operational. It consists of some 100K lines of C++ and Java and runs on both Unix- and Linux-based platforms. It was constructed with the cooperation of students and faculty at Brown, Brandeis, and MIT. The fundamental design of the engine has been well documented elsewhere: the architecture of the engine is described in [9], while the scheduling algorithms are presented in [10]. Load-shedding algorithms are presented in [21], and our approach to high availability in a multi-site Aurora installation is covered in [12, 15]. Lastly, we have been involved in a collective effort to define a benchmark that described the sort of monitoring applications that we have in mind. The result of this effort is called “Linear Road” and is described in [7].

We have used Aurora to build various application systems. The first application we describe here is an Aurora implementation of Linear Road, mentioned above. Second, we have implemented a pilot application that detects late arrival of messages in a financial-services feed-processing environment. Third, one of our collaborators, a military medical research laboratory [23], asked us to build a system to monitor the levels of hazardous materials in fish. Lastly, we have used Aurora to build Medusa [8], a distributed version of Aurora that is intended to be used by multiple enterprises that operate in different administrative domains.

The current Aurora prototype has been transferred to the commercial domain, with venture capital backing. As such, the academic project is hard at work on a complete redesign of Aurora, which we call Borealis. Borealis is a distributed stream-processing system that inherits core stream-processing functionality from Aurora and distribution functionality from Medusa. Borealis modifies and extends both systems in nontrivial and critical ways to provide advanced capabilities that are commonly required by newly emerging stream-processing applications. The Borealis design is driven by our experience in using Aurora and Medusa, in developing several streaming applications including the Linear Road benchmark, and several commercial opportunities. Borealis will address the following requirements of newly emerging streaming applications.

We start with a review of the Aurora design and implementation in Sect. 2. We then present the case studies mentioned above in detail in Sect. 3 and provide a brief retrospective on what we have learned throughout the process in Sect. 4. We conclude in Sect. 5 by briefly discussing the ideas we have for Borealis in several new areas including mechanisms for dynamic modification of query specification and query results and a distributed optimization framework that operates across server and sensor networks.

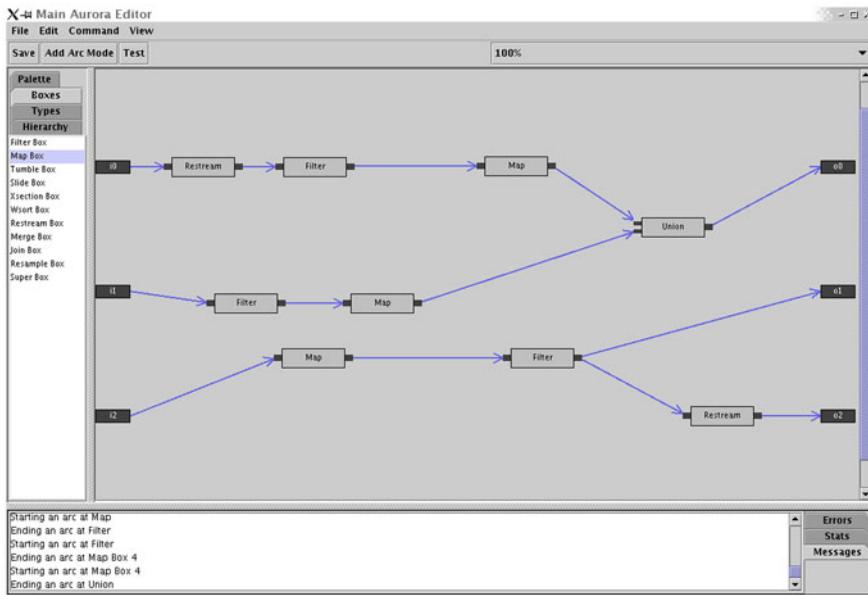


Fig. 1 Aurora graphical user interface

2 The Aurora Centralized Stream Processing Engine

Aurora is based on a dataflow-style “boxes and arrows” paradigm. Unlike other stream processing systems that use SQL-style declarative query interfaces (e.g., STREAM [17]), this approach was chosen because it allows query activity to be interspersed with message processing (e.g., cleaning, correlation, etc.). Systems that only perform the query piece must ping-pong back and forth to an application for the rest of the work, thereby adding to system overhead and latency.

In Aurora, a developer uses the GUI to wire together a network of boxes and arcs that will process streams in a manner that produces the outputs necessary to his or her application. A screen shot of the GUI used to create Aurora networks is shown in Fig. 1: the black boxes indicate input and output streams that connect Aurora with the stream sources and applications, respectively. The other boxes are Aurora operators and the arcs represent data flow among the operators. Users can drag-and-drop operators from the palette on the left and connect them by simply drawing arrows between them. It should be noted that a developer can name a collection of boxes and replace it with a “superbox”. This “macro-definition” mechanism drastically eases the development of big networks.

As illustrated in Fig. 2, the Aurora system is, in fact, a directed sub-graph of a workflow diagram that expresses all simultaneous query computations. We refer to this workflow diagram as the Aurora network. Queries are built from a standard set of well-defined operators (a.k.a. boxes). The arcs denote tuple queues that represent streams. Each box, when scheduled, processes one or more tuples from its input

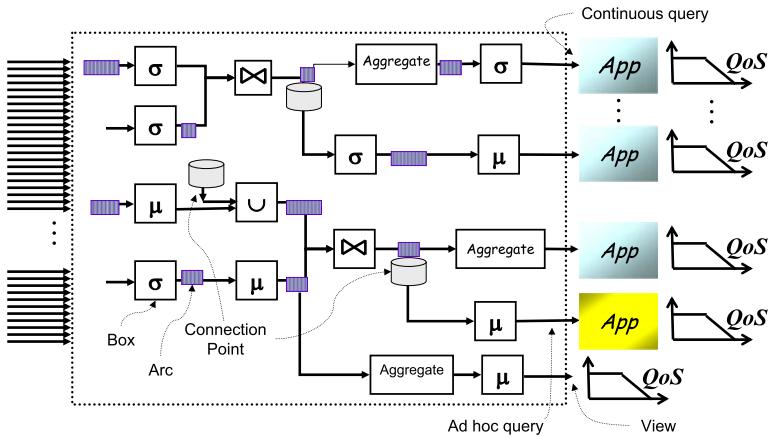


Fig. 2 The Aurora processing network

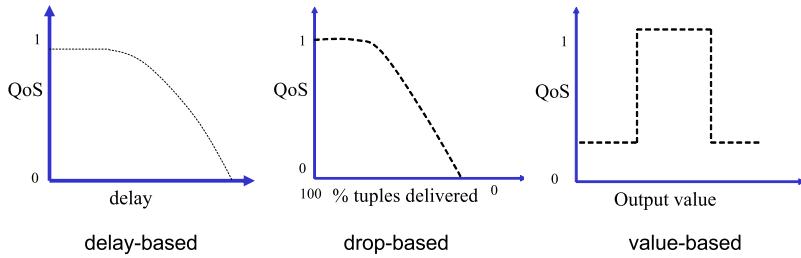
queue and puts the results on its output queue. When tuples are generated at the queues attached to the applications, they are assessed according to the application's QoS specification (more on this below).

By default, queries are continuous in that they can potentially run forever over push-based inputs. Ad hoc queries can also be defined at run time and are attached to connection points, which are predetermined arcs in the network where historical data is stored. Connection points can be associated with persistence specifications that indicate how long a history to keep. Aurora also allows dangling connection points that can store static data sets. As a result, connection points enable Aurora queries to combine traditional pull-based data with live push-based data. Aurora also allows the definition of views, which are queries to which no application is connected. A view is allowed to have a QoS specification as an indication of its importance. Applications can connect to the view whenever there is a need.

The Aurora operators are presented in detail in [5] and are summarized in Fig. 3. Aurora's operator choices were influenced by numerous systems. The basic operators Filter, Map and Union are modeled after the Select, Project and Union operations of the relational algebra. Join's use of a distance metric to relate joinable elements on opposing streams is reminiscent of the relational band join [14]. Aggregate's sliding window semantics is a generalized version of the sliding window constructs of SEQ [19] and SQL-99 (with generalizations including allowance for disorder (SLACK), timeouts, value-based windows etc.). The ASSUME ORDER clause (used in Aggregate and Join), which defines a result in terms of an order which may or may not be manifested, is borrowed from AQuery [16].

Each input must obey a particular schema (a fixed number of fixed or variable length fields of the standard data types). Every output is similarly constrained. An Aurora network accepts inputs, performs message filtering, computation, aggregation, and correlation, and then delivers output messages to applications.

Filter (p_1, \dots, p_m) (S) → Filter → <i>Selection:</i> route tuples to output with 1st predicate match	BSort (Using O) (S) → BSort → <i>Bounded-Pass Bubble Sort:</i> Sort over order attribute. Number of passes = slack	Resample (F, Size s, Left Assume O_1, Right Assume O_2) (S_1, S_2) → Resample → <i>Interpolation:</i> Interpolate missing points in S_2 as determined with S_1
Map ($A_1=f_1, \dots, A_m=f_m$) (S) → Map → <i>Mapping:</i> Map input tuples to outputs with m fields	Aggregate (F, Assume O, Size s, Advance i) (S) → Aggregate → <i>Sliding Window Aggregate:</i> Apply F to windows of size, s . Slide by i between windows.	Read (Assume O, SQL Q) (S) → Read → <i>SQL Read:</i> Apply SQL query Q for each input in S . Output as stream.
Union (S_1, \dots, S_m) → Union → <i>Merging:</i> Merge n input streams into 1	Join (P, Size s, Left Assume O_1, Right Assume O_2) (S_1, S_2) → Join → <i>Band Join:</i> Join s -sized windows of tuples from S_1 and S_2	Update (Assume O, SQL U, Report t) (S) → Update → <i>SQL Write:</i> Apply SQL update query U for each input in S . Report t for each update.

Fig. 3 Aurora operators**Fig. 4** QoS graph types

Moreover, every output is optionally tagged with a Quality of Service (QoS) specification. This specification indicates how much latency the connected application can tolerate, as well as what to do if adequate responsiveness cannot be assured under overload situations. Note that the Aurora notion of QoS is different from the traditional QoS notion that typically implies hard performance guarantees, resource reservations and strict admission control. Specifically, QoS is a multidimensional function of several attributes of an Aurora system. These include (i) response times—output tuples should be produced in a timely fashion; as otherwise QoS will degrade (as delays get longer); (ii) tuple drops—if tuples are dropped to shed load, then the QoS of the affected outputs will deteriorate; and (iii) values produced—QoS clearly depends on whether important values are being produced or not. Figure 4 illustrates these three QoS graph types.

When a developer is satisfied with an Aurora network, he or she can compile it into an intermediate form, which is stored in an embedded database as part of the system catalog. At run-time this data structure is read into virtual memory. The Aurora run-time architecture is shown in Fig. 5. The heart of the system is the scheduler that determines which box (i.e., operator) to run. The scheduler also determines how many input tuples of a box to process and how far to “push” the tuples toward the output. Aurora operators can store and access data from embedded in-memory

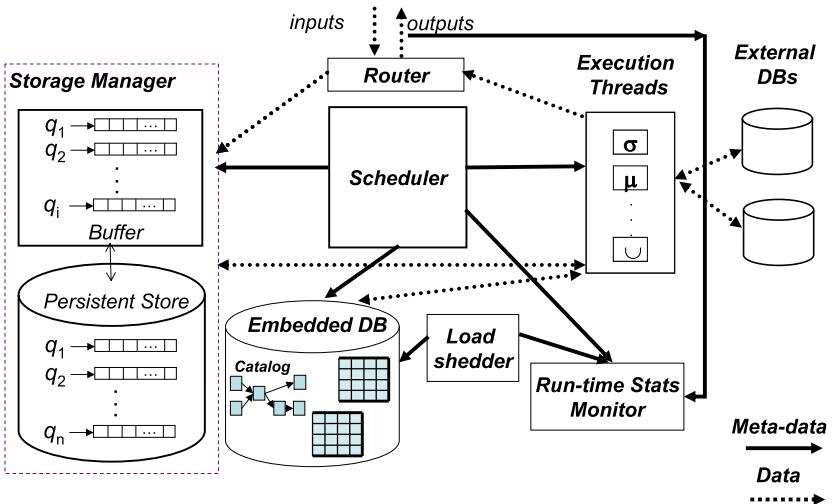


Fig. 5 Aurora run-time architecture

databases as well as from external databases. Aurora also has a Storage Manager that is used to buffer queues when main memory runs out. This is particularly important for queues at connection points since they can grow quite long.

The Run-Time Stats Monitor continuously monitors the QoS of output tuples. This information is important since it drives the Scheduler in its decision-making, and it also informs the Load Shedder when and where it is appropriate to discard tuples in order to shed load. Load shedding is only one of the techniques employed by Aurora to improve the QoS delivered to applications. When load shedding is not working, Aurora will try to re-optimize the network using standard query optimization techniques (such as those that rely on operator commutativities). This tactic requires a more global view of the network and thus is used more sparingly. The final tactic is to retune the scheduler by gathering new statistics or switching scheduler disciplines. The Aurora optimizer can rearrange a network by performing box swapping when it thinks the result will be favorable. Such box swapping cannot occur across a connection point; hence connection points are arcs that restrict the behavior of the optimizer as well as remember history. More detailed information on these various topics can be obtained from the referenced papers [5, 9, 10, 21].

3 Aurora Case Studies

In this section, we present four case studies of applications built using the Aurora engine and tools.

3.1 Financial Services Application

Financial service organizations purchase stock ticker feeds from multiple providers and need to switch in real time between these feeds if they experience too many problems. We worked with a major financial services company on developing an Aurora application that detects feed problems and triggers the switch in real time. In this section, we summarize the application (as specified by the financial services company) and its implementation in Aurora.

An unexpected delay in the reporting of new prices is an example of a feed problem. Each security has an expected reporting interval, and the application needs to raise an alarm if a reporting interval exceeds its expected value. Furthermore, if more than some number of alarms are recorded, a more serious alarm is raised that could indicate that it is time to switch feeds. The delay can be caused by the underlying exchange (e.g., NYSE, NASDAQ) or by the feed provider (e.g., Comstock, Reuters). If it is the former, switching to another provider will not help, so the application must be able to rapidly distinguish between these two cases.

Ticker information is provided as a real-time data feed from one or more providers, and a feed typically reports more than one exchange. As an example, let us assume that there are 500 securities within a feed that update at least once every 5 s and they are called “fast updates”. Let us also assume that there are 4000 securities that update at least once every 60 s and they are called “slow updates”.

If a ticker update is not seen within its update interval, the monitoring system should raise a *low alarm*. For example, if MSFT is expected to update within 5 s, and 5 s or more elapse since the last update, a low alarm is raised.

Since the source of the problem could be in the feed or the exchange, the monitoring application must count the number of low alarms found in each exchange and the number of low alarms found in each feed. If the number for each of these categories exceeds a threshold (100 in the following example), a *high alarm* is raised. The particular high alarm will indicate what action should be taken. When a high alarm is raised, the low alarm count is reset and the counting of low alarms begins again. In this way, the system produces a high alarm for every 100 low alarms of a particular type.

Furthermore, the posting of a high alarm is a serious condition, and low alarms are suppressed when the threshold is reached to avoid distracting the operator with a large number of low alarms.

Figure 6 presents our solution realized with an Aurora query network. We assume for simplicity that the securities within each feed are already separated into the 500 fast updating tickers and the 4000 slowly updating tickers. If this is not the case, then the separation can be easily achieved with a lookup. The query network in Fig. 6 actually represents six different queries (one for each output). Notice that much of the processing is shared.

The core of this application is in the detection of late tickers. Boxes 1, 2, 3, and 4 are all Aggregate boxes that perform the bulk of this computation. An Aggregate box groups input tuples by common value of one or more of their attributes, thus effectively creating a substream for each possible combination of these attribute

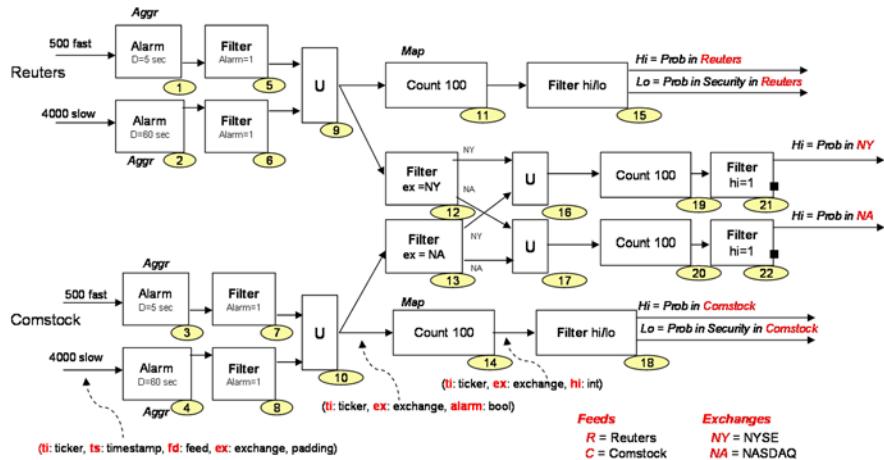


Fig. 6 Aurora query network for the alarm correlation application

values. In this case, the aggregates are grouping the input on common value of ticker symbol. For each grouping or substream, a window is defined that demarcates interesting runs of consecutive tuples called *windows*. For each of the tuples in one of these windows, some memory is allocated and an aggregating function (e.g., Average) is applied. In this example, the window is defined to be every consecutive pair (e.g., tuples 1 and 2, tuples 2 and 3, etc.) and the aggregating function generates one output tuple per window with a boolean flag called *Alarm*, which is a 1 when the second tuple in the pair is delayed (call this an *Alarm tuple*) and a 0 when it is on time.

Aurora's operators have been designed to react to imperfections such as delayed tuples. Thus, the triggering of an Alarm tuple is accomplished directly using this built-in mechanism. The window defined on each pair of tuples will *timeout* if the second tuple does not arrive within the given threshold (5 s in this case). In other words, the operator will produce one alarm each time a new tuple fails to arrive within 5 s, as the corresponding window will automatically timeout and close. The high-level specification of Aggregate boxes 1 through 4 is:

```
Aggregate(Group by ticker,
          Order on arrival,
          Window (Size = 2 tuples,
                  Step = 1 tuple,
                  Timeout = 5 sec))
```

Boxes 5 through 8 are Filters that eliminate the normal outputs, thereby letting only the Alarm tuples through. Box 9 is a Union operator that merges all Reuters alarms onto a single stream. Box 10 performs the same operation for Comstock.

The rest of the network determines when a large number of Alarms is occurring and what the cause of the problem might be. Boxes 11 and 15 count Reuters alarms

and raise a high alarm when a threshold (100) is reached. Until that time, they simply pass through the normal (low) alarms. Boxes 14 and 18 do the same for Comstock. Note that the boxes labeled *Count 100* are actually Map boxes. Map takes a user-defined function as a parameter and applies it to each input tuple. That is, for each tuple t in the input stream, a Map box parameterized by a function f produces the tuple $f(t)$. In this example, *Count 100* simply applies the following user-supplied function (written in pseudocode) to each tuple that passes through:

```

F (x:tuple) = cnt++
if (cnt % 100 == 0)
    if !suppress
        emit lo-alarm
    else
        emit drop-alarm
else
    emit hi-alarm
    set suppress = true

```

Boxes 12, 13, 16, and 17 separate the alarms from both Reuters and Comstock into alarms from NYSE and alarms from NASDAQ. This is achieved by using Filters to take NYSE alarms from both feed sources (Boxes 12 and 13) and merging them using a Union (Box 16). A similar path exists for NASDAQ alarms. The results of each of these streams are counted and filtered as explained above.

In summary, this example illustrates the ability to share computation among queries, the ability to extend functionality through user-defined Aggregate and Map functions, and the need to detect and exploit stream imperfections.

3.2 The Linear Road Benchmark

Linear Road is a benchmark for stream-processing engines [4, 7]. This benchmark simulates an urban highway system that uses “variable tolling” (also known as “congestion pricing”) [1, 13, 18], where tolls are determined according to such dynamic factors as congestion, accident proximity, and travel frequency. As a benchmark, Linear Road specifies input data schemas and workloads, a suite of continuous and historical queries that must be supported, and performance (query and transaction response time) requirements.

Variable tolling is becoming increasingly prevalent in urban settings because it is effective at reducing traffic congestion and because recent advances in microsensor technology make it feasible. Traffic congestion in major metropolitan areas is an increasing problem as expressways cannot be built fast enough to keep traffic flowing freely at peak periods. The idea behind variable tolling is to issue tolls that vary according to time-dependent factors such as congestion levels and accident proximity with the motivation of charging higher tolls during peak traffic periods to discourage

vehicles from using the roads and contributing to the congestion. Illinois, California, and Finland are among the highway systems that have pilot programs utilizing this concept.

The benchmark itself assumes a fictional metropolitan area (called “Linear City”) that consists of 10 expressways of 100-mile-long segments each and 1,000,000 vehicles that report their positions via GPS-based sensors every 30 s. Tolls must be issued on a per-segment basis automatically, based on statistics gathered over the previous 5 min concerning average speed and number of reporting cars. A segment’s tolls are overridden when accidents are detected in the vicinity (an accident is detected when multiple cars report close positions at the same time), and vehicles that use a particular expressway often are issued “frequent traveler” discounts.

The Linear Road benchmark demands support for five queries: two continuous and three historical. The first continuous query calculates and reports a segment toll every time a vehicle enters a segment. This toll must then be charged to the vehicle’s account when the vehicle exits that segment without exiting the expressway. Again, tolls are based on current congestion conditions on the segment, recent accidents in the vicinity, and frequency of use of the expressway for the given vehicle. The second continuous query involves detecting and reporting accidents and adjusting tolls accordingly. The historical queries involve requesting an account balance or a day’s total expenditure for a given vehicle on a given expressway and a prediction of travel time between two segments on the basis of average speeds on the segments recorded previously. Each of the queries must be answered with a specified accuracy and within a specified response time. The degree of success for this benchmark is measured in terms of the number of expressways the system can support, assuming 1000 position reports issued per second per expressway, while answering each of the five queries within the specified latency bounds.

3.3 Environmental Monitoring

We have also worked with a military medical research laboratory on an application that involves monitoring toxins in the water. This application is fed streams of data indicating fish behavior (e.g., breathing rate) and water quality (e.g., temperature, pH, oxygenation, and conductivity). When the fish behave abnormally, an alarm is sounded.

Input data streams were supplied by the army laboratory as a text file. The single data file interleaved fish observations with water quality observations. The alarm message emitted by Aurora contains fields describing the fish behavior and two different water quality reports: the water quality at the time the alarm occurred and the water quality from the last time the fish behaved normally. The water quality reports contain not only the simple measurements but also the 1-/2-/4-hour sliding-window deltas for those values.

The application’s Aurora processing network is shown in Fig. 7 (snapshot taken from the Aurora GUI): The input port (1) shows where tuples enter Aurora from

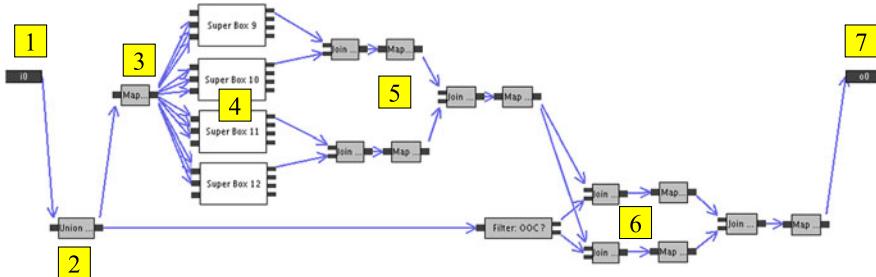


Fig. 7 Aurora query network for the environmental contamination detection applications (GUI snapshot)

the outside data source. In this case, it is the application's C++ program that reads in the sensor log file. A Union box (2) serves merely to split the stream into two identical streams. A Map box (3) eliminates all tuple fields except those related to water quality. Each superbox (4) calculates the sliding-window statistics for one of the water quality attributes. The parallel paths (5) form a binary join network that brings the results of (4)'s subnetworks back into a single stream. The top branch in (6) has all the tuples where the fish act oddly, and the bottom branch has the tuples where the fish act normally. For each of the tuples sent into (1) describing abnormal fish behavior, (6) emits an alarm message tuple. This output tuple has the sliding-window water quality statistics for both the moment the fish acted oddly and for the most recent previous moment that the fish acted normally. Finally, the output port (7) shows where result tuples are made available to the C++-based monitoring application. Overall, the entire application ended up consisting of 3400 lines of C++ code (primarily for file parsing and a simple monitoring GUI) and a 53-operator Aurora query network.

During the development of the application, we observed that Aurora's stream model proved very convenient for describing the required sliding-window calculations. For example, a single instance of the aggregate operator computed the 4-h sliding-window deltas of water temperature.

Aurora's GUI for designing query networks also proved invaluable. As the query network grew large in the number of operators used, there was great potential for overwhelming complexity. The ability to manually place the operators and arcs on a workspace, however, permitted a visual representation of "subroutine" boundaries that let us comprehend the entire query network as we refined it.

We found that small changes in the operator language design would have greatly reduced our processing network complexity. For example, Aggregate boxes apply some window function [such as $\text{DELTA}(\text{water-pH})$] to the tuples in a sliding window. Had an Aggregate box been capable of evaluating multiple functions at the same time on a single window [such as $\text{DELTA}(\text{water-pH})$ and $\text{DELTA}(\text{watertemp})$], we could have used significantly fewer boxes. Many of these changes have since been made to Aurora's operator language.

The ease with which the processing flow could be experimentally reconfigured during development, while remaining comprehensible, was surprising. It appears

Table 1 Overview of a subset of the Aurora API

<code>start</code> and <code>shutdown</code> : Respectively starts processing and shuts down a complete query network.
<code>modifyNetwork</code> : At runtime, adds or removes schemas, streams, and operator boxes from a query network processed by a single Aurora engine.
<code>typecheck</code> : Validates (part of) a query network. Computes properties of intermediate and output streams.
<code>enqueue</code> and <code>dequeue</code> : Push and pull tuples on named streams.
<code>listEntities</code> and <code>describe(Entity)</code> : Provide information on entities in the current query network.
<code>getPerfStats</code> : Provides performance and load information.

that this was only possible by having both a well-suited operator set and a GUI tool that let us visualize the processing. It seems likely that this application was developed at least as quickly in Aurora as it would have been with standard procedural programming.

We note that, for this particular application, real-time response was not required. The main value Aurora added in this case was the ease of developing stream-oriented applications.

3.4 Medusa: Distributed Stream Processing

Medusa is a distributed stream-processing system built using Aurora as the single-site query-processing engine. Medusa takes Aurora queries and distributes them across multiple nodes. These nodes can all be under the control of one entity or be organized as a loosely coupled federation under the control of different autonomous participants.

A distributed stream-processing system such as Medusa offers several benefits including incremental scalability over multiple nodes, composition of stream feeds across multiple participants, and high availability and load sharing through resource multiplexing.

The development of Medusa prompted two important changes to the Aurora processing engine. First, it became apparent that it would be useful to offer Aurora not only as a stand-alone system but also as a library that could easily be integrated within a larger system. Second, we felt the need for an Aurora API, summarized in Table 1. This API is composed of three types of methods: (i) methods to set up queries and push or pull tuples from Aurora, (ii) methods to modify query networks at runtime (operator additions and removals), and (iii) methods giving access to performance information.

4 Experience and Lessons Learned

4.1 Support for Historical Data

From our work on a variety of streaming applications, it became apparent that each application required maintaining and accessing a collection of historical data. For example, the Linear Road benchmark, which represents a realistic application, required maintaining 10 weeks of toll history for each driver, as well as the current positions of every vehicle and the locations of accidents tying up traffic. Historical data might be used to support *historical queries* (e.g., tell me how much driver X has spent on tolls on expressway Y over the past 10 weeks) or serve as inputs to *hybrid queries* involving both streaming and historical data [e.g., report the current toll for vehicle X based on its current position (streamed data) and the presence of any accidents in its vicinity (historical data)].

In the applications we have looked at, historical data take three different forms. These forms differ by their *update patterns*—the means by which incoming stream data are used to update the contents of a historical collection. These forms are summarized below.

1. **Open windows (connection points).** Linear Road requires maintaining the *last 10 weeks' worth of toll data for each driver* to support both historical queries and integrated queries. This form of historical data resembles a window in its FIFO-based update pattern but must be shared by multiple queries and therefore be openly accessible.
2. **Aggregate summaries (latches).** Linear Road requires maintaining such aggregated historical data as: the current toll balance for every vehicle ($\text{SUM}(\text{Toll})$), the last reported position of every vehicle ($\text{MAX}(\text{Time})$), and the average speed on a given segment over the past 5 min ($\text{AVG}(\text{Speed})$). In all cases, the update patterns involve maintaining data by key value (e.g., vehicle or segment ID) and using incoming tuples to update the aggregate value that has the appropriate key. As with open windows, aggregate summaries must be shared by multiple queries and therefore must be openly accessible.
3. **Tables.** Linear Road requires maintaining tables of historical data whose update patterns are arbitrary and determined by the values of streaming data. For example, a table must be maintained that holds every accident that has yet to be cleared (such that an accident is detected when multiple vehicles report the same position at the same time). This table is used to determine tolls for segments in the vicinity of the accident and to alert drivers approaching the scene of the accident. The update pattern for this table resembles neither an open window nor an aggregate summary. Rather, accidents must be deleted from the table when an incoming tuple reports that the accident has been cleared. This requires the declaration of an arbitrary update pattern.

Whereas open windows and aggregate summaries have fixed update patterns, tables require update patterns to be explicitly specified. Therefore, the Aurora query

algebra (SQuAl) includes an Update box that permits an update pattern to be specified in SQL. This box has the form

$$\text{UPDATE (Assume } O, \text{ SQL } U, \text{ Report } t\text{)}$$

such that U is an SQL update issued with every incoming tuple and includes variables that get instantiated with the values contained in that tuple. O specifies the assumed ordering of input tuples, and t specifies a tuple to output whenever an update takes place. Further, because all three forms of historical collections require random access, SQuAl also includes a Read box that initiates a query over stored data (also specified in SQL) and returns the result as a stream. This box has the form

$$\text{READ (Assume } O, \text{ SQL } Q\text{)}$$

such that Q is an SQL query issued with every incoming tuple and includes variables that get instantiated with the values contained in that tuple.

4.2 Synchronization

As continuous queries, stream applications inherently rely on shared data and computation. Shared data might be contained in a table that one query updates and another query reads. For example, the Linear Road application requires that vehicle position data be used to update statistics on highway usage, which in turn are read to determine tolls for each segment on the highway. Alternatively, box output can be shared by multiple queries to exploit common subexpressions or even by a single query as a way of merging intermediate computations after parallelization.

Transactions are required in traditional databases because data sharing can lead to data inconsistencies. An equivalent synchronization mechanism is required in streaming settings, as data sharing in this setting can also lead to inconsistencies. For example, if a toll charge can expire, then a toll assessment to a given vehicle should be delayed until a new toll charge is determined. The need for synchronization with data sharing is achieved in SQuAl via the WaitFor box whose syntax is shown below:

$$\text{WaitFor (P: Predicate, T: Timeout).}$$

This binary operator buffers each tuple t on one input stream until a tuple arrives on the second input stream that with t satisfies P (or until the timeout expires, in which case t is discarded). If a Read operation must follow a given Update operation, then a WaitFor can buffer the Read request (tuple) until a tuple output by the Update box (and input to the second input of WaitFor) indicates that the Read operation can proceed.

4.3 Resilience to Unpredictable Stream Behavior

Streams are by their nature unpredictable. Monitoring applications require the system to continue operation even when the unpredictable happens. Sometimes, the

only way to do this is to produce approximate answers. Obviously, in these cases, the system should try to minimize errors.

We have seen examples of streams that do not behave as expected. The financial services application that we described earlier requires the ability to detect a problem in the arrival rate of a stream. The military application must fundamentally adjust its processing to fit the available resources during times of stress. In both of these cases, Aurora primitives for unpredictable stream behavior were brought to bear on the problem.

Aurora makes no assumptions that a data stream arrives in any particular order or with any temporal regularity. Tuples can be late or out of order due to the nature of the data sources, the network that carries the streams, or the behavior of the operators themselves. Accordingly, our operator set includes user-specified parameters that allow handling such “damaged” streams gracefully.

For many of the operators, an input stream can be specified to obey an expected order. If out-of-order data are known to the network designer not to be of relevance, the operator will simply drop such data tuples immediately. Nonetheless, Aurora understands that this may at times be too drastic a constraint and provides an optional slack parameter to allow for some tolerance in the number of data tuples that may arrive out of order. A tuple that arrives out of order within the slack bounds will be processed as if it had arrived in order.

With respect to possible irregularity in the arrival rate of data streams, the Aurora operator set offers all windowed operators an optional timeout parameter. The timeout parameter tells the operator how long to wait for the next data tuple to arrive. This has two benefits: it prevents blocking (i.e., no output) when one stream is stalled, and it offers another way for the network designer to characterize the value of data that arrive later than they should, as in the financial services application in which the timeout parameter was used to determine when a particular data packet arrived late.

4.4 XML and Other Feed Formats Adaptor Required

Aurora provides a network protocol that may be used to enqueue and dequeue tuples via Unix or TCP sockets. The protocol is intentionally very low-level: to eliminate copies and improve throughput, the tuple format is closely tied to the format of Aurora’s internal queue format. For instance, the protocol requires that each packet contain a fixed amount of padding reserved for bookkeeping and that integer and floating-point fields in the packet match the architecture’s native format.

While we anticipate that performance-critical applications will use our low-level protocol, we also recognize that the formats of Aurora’s input streams may be outside the immediate control of the Aurora user or administrator, for example, stock quote data arriving in XML format from a third-party information source. Also, even if the streams are being generated or consumed by an application within an organization’s control, in some cases protocol stability and portability (e.g., not requiring

the client to be aware of the endianness of the server architecture) are important enough to justify a minor performance loss.

One approach to addressing these concerns is to simply require the user to build a proxy application that accepts tuples in the appropriate format, converts them to Aurora’s internal format, and pipes them into the Aurora process. This approach, while simple, conflicts with one of Aurora’s key design goals—to minimize the number of boundary crossings in the system—since the proxy application would be external to Aurora and hence live in its own address space.

We resolve this problem by allowing the user to provide plug-ins called *converter boxes*. Converter boxes are shared libraries that are dynamically linked into the Aurora process space; hence their use incurs no boundary crossings. A user-defined *input converter box* provides a hook that is invoked when data arrive over the network. The implementation may examine the data and inject tuples into the appropriate streams in the Aurora network. This may be as simple as consuming fixed-length packets and enforcing the correct byte order on fields or as complex as transforming fully formed XML documents into tuples. An *output converter box* performs the inverse function: it accepts tuples from streams in Aurora’s internal format and converts them into a byte stream to be consumed by an external application.

Input and output converter boxes are powerful connectivity mechanisms: they provide a high level of flexibility in dealing with external feeds and sinks without incurring a performance hit. This combination of flexibility and high performance is essential in a streaming database that must assimilate data from a wide variety of sources.

4.5 Programmatic Interfaces and Globally Accessible Catalogs Are a Good Idea

Initially, Aurora networks were created using the GUI and all Aurora metadata (i.e., catalogs) were stored in an internal representation. Our experience with the Medusa system quickly made us realize that, in order for Aurora to be easily integrated within a larger system, a higher-level, *programmatic interface* was needed to script Aurora networks and metadata needed to be globally accessible and updatable.

Although we initially assumed that only Aurora itself (i.e., the runtime and the GUI) would need direct access to the catalog representation, we encountered several situations where this assumption did not hold. For instance, in order to manage distribution operation across multiple Aurora nodes, Medusa required knowledge of the contents of node catalogs and the ability to selectively move parts of catalogs from node to node. Medusa needed to be able to create catalog objects (schema, streams, and boxes) without direct access to the Aurora catalog database, which would have violated abstraction. In other words, relying on the Aurora runtime and GUI as the sole software components able to examine and modify catalog structures turned out to be an unworkable solution when we tried to build sophisticated applications on

the Aurora platform. We concluded that we needed a simple and transparent catalog representation that is easily readable and writable by external applications. This would make it much easier to write higher-level systems that use Aurora (such as Medusa) and alternative authoring tools for catalogs.

To this end, Aurora currently incorporates appropriate interfaces and mechanisms (Sect. 3.4) to make it easy to develop external applications to inspect and modify Aurora query networks. A universally readable and writable catalog representation is crucial in an environment where multiple applications may operate on Aurora catalogs.

4.6 Performance Critical

Fundamental to an SPE is a high-performance “message bus”. This is the system that moves tuples from one operator to the next, storing them temporarily, as well as into and out of the query network. Since every tuple is passed on the bus a number of times, this is definitely a performance bottleneck. Even such trivial optimizations as choosing the right `memcpy()` implementation gave substantial improvements to the whole system.

Second to the message bus, the scheduler is the core element of an SPE. The scheduler is responsible for allocating processor time to operators. It is tempting to decorate the scheduler with all sorts of high-level optimization such as intelligent allocation of processor time or real-time profiling of query plans. But it is important to remember that scheduler overhead can be substantial in networks where there are many operators and that the scheduler makes no contribution to the actual processing. All addition of scheduler functionality must be greeted with skepticism and should be aggressively profiled.

Once the core of the engine has been aggressively optimized, the remaining hot spots for performance are to be found in the implementation of the operators. In our implementation, each operator has a “tight loop” that processes batches of input tuples. This loop is a prime target for optimization. We make sure nothing other than necessary processing occurs in the loop. In particular, housekeeping of data structures such as memory allocations and deallocation needs to be done outside of this loop so that its cost can be amortized across many tuples.

Data structures are another opportunity for operator optimization. Many of our operators are stateful; they retain information or even copies of previous input. Because these operators are asked to process and store large numbers of tuples, efficiency of these data structures is important. Ideally, processing of each input tuple is accomplished in constant time. In our experience, processing that is linear in the amount of states stored is unacceptable.

In addition to the operators themselves, any parts of the system that are used by those operators in the tight loops must be carefully examined. For example, we have a small language used to specify expressions for Map operators. Because these expressions are evaluated in such tight loops, optimizing them was important. The addition of an expensive compilation step may even be appropriate.

These microbenchmarks measure the overhead involved in passing tuples into and out of Aurora boxes and networks; they do not measure the time spent in boxes performing nontrivial operations such as joining and aggregation. Message-passing overhead, however, can be a significant time sink in streaming databases (as it was in earlier versions of Aurora). Microbenchmarking was very useful in eliminating performance bottlenecks in Aurora’s message-passing infrastructure. This infrastructure is now fast enough in Aurora that nontrivial box operations are the only noticeable bottleneck, i.e., CPU time is overwhelmingly devoted to useful work and not simply to shuffling around tuples.

5 Ongoing Work: The Borealis Distributed SPE

This section presents the initial ideas that we have started to explore in the context of the Borealis distributed SPE, which is a follow-on to Aurora. The rest of the section will provide an overview of the new challenges that Borealis will address. More details on these challenges as well as a preliminary design of Borealis can be found in [2].

5.1 Dynamic Revision of Query Results

In many real-world streams, corrections or updates to previously processed data are available only after the fact. For instance, many popular data streams, such as the Reuters stock market feed, often include messages that allow the feed originator to correct errors in previously reported data. Furthermore, stream sources (such as sensors), as well as their connectivity, can be highly volatile and unpredictable. As a result, data may arrive late and miss their processing window or be ignored temporarily due to an overload situation. In all these cases, applications are forced to live with imperfect results, unless the system has means to correct its processing and results to take into account newly available data or updates.

The Borealis data model extends that of Aurora by supporting such corrections by way of revision records. The goal is to process revisions intelligently, correcting query results that have already been emitted in a manner that is consistent with the corrected data. Processing of a revision message must replay a portion of the past with a new or modified value. Thus, to process revision messages correctly, we must make a query diagram “replayable”. In theory, we could process each revision message by replaying processing from the point of the revision to the present. In most cases, however, revisions on the input affect only a limited subset of output tuples, and to regenerate unaffected output is wasteful and unnecessary. To minimize runtime overhead and message proliferation, we assume a closed model for replay that generates revision messages when processing revision messages. In other words, our model processes and generates “deltas” showing only the effects of revisions rather than regenerating the entire result. The primary challenge here is to develop efficient revision-processing techniques that can work with bounded history.

5.2 Dynamic Query Modification

In many stream-processing applications, it is desirable to change certain attributes of the query at runtime. For example, in the financial services domain, traders typically wish to be alerted of *interesting* events, where the definition of “interesting” (i.e., the corresponding filter predicate) varies based on current context and results. In network monitoring, the system may want to obtain more precise results on a specific subnetwork if there are signs of a potential denial-of-service attack. Finally, in a military stream application that MITRE [23] explained to us, they wish to switch to a “cheaper” query when the system is overloaded. For the first two applications, it is sufficient to simply alter the operator parameters (e.g., window size, filter predicate), whereas the last one calls for altering the operators that compose the running query. Another motivating application comes again from the financial services community. Universally, people working on trading engines wish to test out new trading strategies as well as debug their applications on historical data before they go live. As such, they wish to perform “time travel” on input streams. Although this last example can be supported in most current SPE prototypes (i.e., by attaching the engine to previously stored data), a more user-friendly and efficient solution would obviously be desirable.

Two important features that will facilitate online modification of continuous queries in Borealis are *control lines* and *time travel*. Control lines extend Aurora’s basic query model with the ability to change operator parameters as well as operators themselves on the fly. Control lines carry messages with revised box parameters and new box functions. For example, a control message to a Filter box can contain a reference to a boolean-valued function to replace its predicate. Similarly, a control message to an Aggregate box may contain a revised window size parameter. Additionally, each control message must indicate when the change in box semantics should take effect. Change is triggered when a monotonically increasing attribute received on the data line attains a certain value. Hence, control messages specify an \langle attribute, value \rangle pair for this purpose. For windowed operators like Aggregate, control messages must also contain a flag to indicate if open windows at the time of change must be prematurely closed for a clean start.

Time travel allows multiple queries (different queries or versions of the same query) to be easily defined and executed concurrently, starting from different points in the past or “future” (typically by running a simulation of some sort). To support these capabilities, we leverage three advanced mechanisms in Borealis: enhanced connection points, connection point versions, and revision messages. To facilitate time travel, we define two new operations on connection points. The *replay operation* replays messages stored at a connection point from an arbitrary message in the past. The *offset operation* is used to set the connection point offset in time. When offset into the past, a connection point delays current messages before pushing them downstream. When offset into the future, the connection point predicts future data. When producing future data, various prediction algorithms can be used based on the application. A connection point version is a distinctly named logical copy of a connection point. Each named version can be manipulated independently. It is possible

to shift a connection point version backward and forward in time without affecting other versions.

To replay history from a previous point in time t , we use revision messages. When a connection point receives a replay command, it first generates a set of revision messages that delete all the messages and revisions that have occurred since t . To avoid the overhead of transmitting one revision per deleted message, we use a macro message that summarizes all deletions. Once all messages are deleted, the connection point produces a series of revisions that insert the messages and possibly their following revisions back into the stream. During replay, all messages and revisions received by the connection point are buffered and processed only after the replay terminates, thus ensuring that simultaneous replays on any path in the query diagram are processed in sequence and do not conflict. When offset into the future, time-offset operators predict future values. As new data become available, these predictors can (but do not have to) produce more accurate revisions to their past predictions. Additionally, when a predictor receives revision messages, possibly due to time travel into the past, it can also revise its previous predictions.

5.3 Distributed Optimization

Currently, commercial stream-processing applications are popular in industrial process control (e.g., monitoring oil refineries and cereal plants), financial services (e.g., feed processing, trading engine support and compliance), and network monitoring (e.g., intrusion detection, fraud detection). Here we see a *server-heavy* optimization problem—the key challenge is to process high-volume data streams on a collection of resource-rich “beefy” servers. Over the horizon, we see a very large number of applications of wireless sensor technology (e.g., RFID in retail applications, cell phone services). Here we see a *sensor-heavy* optimization problem—the key challenges revolve around extracting and processing sensor data from a network of resource-constrained “tiny” devices. Further over the horizon, we expect sensor networks to become faster and increase in processing power. In this case the optimization problem becomes more balanced, becoming *sensor-heavy/server-heavy*. To date, systems have exclusively focused on either a server-heavy environment or a sensor-heavy environment. Off into the future, there will be a need for a more flexible optimization structure that can deal with a very large number of devices and perform cross-network sensor-heavy/server-heavy resource management and optimization.

The purpose of the Borealis optimizer is threefold. First, it is intended to optimize processing across a combined sensor and server network. To the best of our knowledge, no previous work has studied such a cross-network optimization problem. Second, QoS is a metric that is important in stream-based applications, and optimization must deal with this issue. Third, scalability, sizewise and geographical, is becoming a significant design consideration with the proliferation of stream-based applications that deal with large volumes of data generated by multiple distributed

sensor networks. As a result, Borealis faces a unique, multiresource/multimetric optimization challenge that is significantly different than the optimization problems explored in the past. Our current thinking is that Borealis will rely on a hierarchical, distributed optimizer that runs at different time granularities [3].

5.4 High Availability

Another part of the Borealis vision involves addressing recovery and high-availability issues. High availability demands that node failure be masked by seamless handoff of processing to an alternate node. This is complicated by the fact that the optimizer will dynamically redistribute processing, making it more difficult to keep backup nodes synchronized. Furthermore, wide-area Borealis applications are not only vulnerable to node failures but also to network failures and more importantly to network partitions. We have preliminary research in this area that leverages Borealis mechanisms including connection point versions, revision tuples, and time travel.

5.5 Implementation Status

We built a Borealis prototype on top of the Aurora and Medusa code bases. Borealis borrowed many of the Aurora modules including its GUI, the XML representation for query diagrams, portions of the runtime system, and much of the logic for boxes. Borealis also borrowed basic networking and distribution logic from Medusa.

The Borealis prototype was demonstrated in SIGMOD 2006 [6], running real-time player-visualization queries on top of a multiplayer network game. The prototype systems is also available to public through the Borealis website [22].

5.6 Commercialization

The Aurora/Borealis project led to the first commercial real-time stream processing engine, which is being developed and offered by StreamBase Inc. [20]. The company was founded in 2003 primarily by the members of the academic project. Since then, StreamBase has grown to more than 60 employees (as of summer 2007) and has a diverse client base that consists of financial services, telecommunications and gaming companies, as well as the intelligence and military sector.

The company has been actively participating in the development and publicity of StreamSQL, an extension of SQL for live data streams, as the standard textual language to develop stream-oriented applications. StreamSQL has constructs that can seamlessly mix streams and stored tables in a single query and expressive pattern matching capabilities.

Acknowledgements This work was supported in part by the National Science Foundation under the grants IIS-0086057, IIS-0325525, IIS-0325703, and IIS-0325838; and by the Army contract DAMD17-02-2-0048. We would like to thank all past members of the Aurora, Medusa, and Borealis projects for their valuable contributions.

References

1. A guide for hot lane development: a US department of transportation federal highway administration. <http://www.itsdocs.fhwa.dot.gov/JPODOCS/REPTSTE/13668.html>
2. D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, J. Janotti, W. Lindner, S. Madden, A. Rasin, M. Stonebraker, N. Tatbul, Y. Xing, S. Zdonik, The design of the Borealis stream processing engine. Technical report CS-04-08, Department of Computer Science, Brown University (2004)
3. D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Rrvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the Borealis stream processing engine, in *CIDR Conference* (2005)
4. D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, S. Zdonik, Aurora: a data stream management system (demo description), in *ACM SIGMOD Conference* (2003)
5. D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S.Z. Aurora, A new model and architecture for data stream management. *VLDB J.* **12**(2) (2003)
6. Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, J. Hwang, A. Jhingran, A. Maskey, O. Pappaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, S. Zdonik, Distributed operation in the Borealis stream processing engine (demo description), in *ACM SIGMOD Conference* (2005)
7. A. Arasu, M. Cherniack, E.F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, Linear road: a stream data management benchmark, in *VLDB* (2004), pp. 480–491
8. M. Balazinska, H. Balakrishnan, M. Stonebraker, Contract-based load management in federated distributed systems, in *NSDI Symposium* (2004)
9. D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik, Monitoring streams—a new class of data management applications, in *VLDB Conference*, Hong Kong, China (2002)
10. D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, M. Stonebraker, Operator scheduling in a data stream manager, in *VLDB Conference*, Berlin, Germany (2003)
11. S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah, TelegraphCQ: continuous dataflow processing for an uncertain world, in *CIDR Conference* (2003)
12. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, S. Zdonik, Scalable distributed stream processing, in *CIDR Conference*, Asilomar, CA (2003)
13. Congestion pricing: a report from intelligent transportation systems (ITS). <http://www.path.berkeley.edu/leap/TTM/DemandManage/pricing.html>
14. D. DeWitt, J. Naughton, D. Schneider, An evaluation of non-equijoin algorithms, in *VLDB Conference*, Barcelona, Catalonia, Spain (1991)
15. J. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, S. Zdonik, A comparison of stream-oriented high-availability algorithms. Technical report CS-03-17, Department of Computer Science, Brown University (2003)
16. A. Lerner, D. Shasha, AQuery: query language for ordered data, optimization techniques, and experiments, in *VLDB Conference*, Berlin, Germany (2003)
17. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma, Query processing, approximation, and resource management in a data stream management system, in *CIDR Conference* (2003)

18. R.W. Poole, Hot lanes prompted by federal program. <http://www.rppi.org/federalhotlanes.html>
19. P. Seshadri, M. Livny, R. Ramakrishnan, SEQ: a model for sequence databases, in *IEEE ICDE Conference*, Taipei, Taiwan (1995)
20. StreamBase incorporated. <http://www.streambase.com/>
21. N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker, Load shedding in a data stream manager, in *VLDB Conference*, Berlin, Germany (2003)
22. The Borealis project web site. <http://www.cs.brown.edu/research/borealis>
23. The MITRE corporation. <http://www.mitre.org/>

Extending Relational Query Languages for Data Streams

**N. Laptev, B. Mozafari, H. Mousavi, H. Thakkar, H. Wang, K. Zeng,
and Carlo Zaniolo**

The design of continuous query languages for data streams and the extent to which these should rely on database query languages represent pivotal issues for data stream management systems (DSMSs). The Expressive Stream Language (ESL) of our Stream Mill system is designed to maximize the spectrum of applications a DSMS can support efficiently, while retaining compatibility with the SQL:2003 standards. This approach offers significant advantages, particularly for the many applications that span both data streams and databases. Therefore, ESL supports minimal extensions required to overcome SQL's expressive power limitations—a critical enhancement since said limitations are quite severe on database applications and are further exacerbated on data stream applications, where, e.g., only nonblocking query

N. Laptev · B. Mozafari · H. Mousavi · K. Zeng · C. Zaniolo (✉)
Computer Science Department, University of California, Los Angeles, CA 90095, USA
e-mail: zaniolo@cs.ucla.edu

N. Laptev
e-mail: nlaptev@cs.ucla.edu

B. Mozafari
e-mail: barzan@cs.ucla.edu

H. Mousavi
e-mail: hmousavi@cs.ucla.edu

K. Zeng
e-mail: kzeng@cs.ucla.edu

H. Thakkar
Google Inc., Mountain View, CA 94043, USA
e-mail: hmt007@gmail.com

H. Wang
Sigma Center, Microsoft Research Asia, Beijing 100190, P.R. China
e-mail: haixun.wang@microsoft.com

operators can be used. Thus, ESL builds on user-defined aggregates and flexible window mechanisms to turn SQL into a powerful and computationally-complete query language, which is capable of supporting applications, such as data stream mining and sequence queries that are beyond the application scope of other DSMSs.

1 Introduction

A key research issue for Data Stream Management Systems (DSMSs) is deciding which data model and query language should be used. A wide spectrum of different solutions have in fact been proposed, including operator-based graphical interfaces [1], programming language extensions [2], and an assortment of other solutions provided in publish/subscribe systems [3]. However, the approach of choice for many research projects is to extend SQL and use this tested database workhorse for continuous queries on data streams. Indeed, an SQL-based approach can offer significant benefits, particularly for the many applications that span both data streams and databases. This is because application developers can then use the same language on both streaming data and stored data—rather than having to learn a new language and cope with the resulting impedance mismatch. From this vantage point, it is also clear that data stream constructs should adhere closely to the syntax and semantics of current standards (namely SQL:2003 [4]); indeed the introduction of constructs that are superficially similar to those of SQL, but actually have different syntax or semantics might confuse, rather than help, users writing spanning applications. Therefore, we advocate a conservative and minimalist’s approach, which limits SQL extensions to those demanded by the very nature of infinite data streams, and the online, push-based computation model of continuous queries. Indeed Data Stream Management Systems (DSMSs) must operate as follows:

- (a) Results must be pushed to the output promptly and eagerly, while input tuples continue to arrive—i.e., without waiting for (i) the results to be requested by other applications (e.g., a procedural program embedding the continuous query), and (ii) the end of input streams (when blocking operators can be finally applied).
- (b) Because of the unbounded and massive nature of the data streams, all past tuples cannot be memorized for future uses. Only synopses, such as windows, can be kept in memory and the rest must be discarded.

The main problem created by (a) is a significant loss of expressive power. Database researchers have long been aware of expressive power limitations of SQL and other relational languages; in fact, this problem has motivated much research on topics such as recursive queries, database mining queries, sequence queries, and time-series queries. Unfortunately, these limitations are dramatically more marring on data stream applications for the following reasons:

1. Blocking query operators that were widely used on databases can no longer be allowed on data streams [5, 6],

2. Database extenders (i.e., libraries of external functions written in procedural languages using BLOBs and CLOBs to exchange data) that have successfully enhanced the versatility of Object Relational (OR) DBMS are much less effective in DSMS, which process data at small increments rather than as aggregate large objects,
3. Embedding queries in a procedural programming language, a solution used extensively in relational database applications, is now of very limited effectiveness on data streams.

We will now expand on these statements starting from point 1. A blocking query operator is one that only returns answers when it detects the end of its input, while a nonblocking operator produces its answers incrementally as new input tuples arrive [5]. For continuous queries, the users must see the results immediately and incrementally as new stream records arrive, rather than when the stream eventually ends: thus only nonblocking query operators are allowed on data streams [5, 6].

The nature of nonblocking queries was formally characterized in [6], where it was shown that only monotonic queries¹ can be expressed by nonblocking operators [6]. Database query languages contain many nonmonotonic (and therefore blocking) query operators/constructs: for instance, set difference and division are nonmonotonic operators in relational algebra, while constructs such as EXCEPT, NOT EXIST, and traditional aggregates in SQL are nonmonotonic. Then, a natural question is whether all the monotonic queries expressible in a query language can be expressed using only its monotonic operators or constructs. A query language that satisfies this criterion is said to be *NB-complete*—i.e., complete for nonblocking queries [6]. A query language that is NB-complete is as suitable a query language for data streams as it is for databases (since by disallowing its nonmonotonic operators/constructs we only lose queries that are of not suitable for continuous online answering). Unfortunately, both relational algebra and SQL-2 fail the NB-completeness test [6]. Thus, the banishment of blocking operators and lack of NB-completeness further aggravate the expressive power limitations of SQL that has made it difficult for DBMS to support new application domains. This problem is discuss next.

As outlined in point 2 above, following the introduction of a time-series datablades by Illustra [7], OR DBMSs offered a plethora of such libraries covering a wide spectrum of applications under an assortment of different names. These functions often use large objects (BLOBs and CLOBs) to exchange data with SQL: for instance, a whole sequence could be encoded as a BLOB and shared between the database and the datablade. This solution is less suitable for data streams, where the computation must proceed continuously in small increments—e.g., by processing each new tuple in the sequence, rather than having to wait for it to be assembled into a BLOB. Indeed, unlike in OR DBMS, datablades have not played an important role as an extension mechanism for DSMS.

¹Queries are viewed as mappings from the database, or the data stream, to the query answer.

Point 3 above considers the solution of embedding SQL queries in a procedural programming language (PL); currently this represents a commonly used approach for developing complex database applications, since the application logic that cannot be expressed in SQL can then be easily implemented in the PL. This solution, however, uses cursors and get-next constructs; thus it relies on a pull-based computation model that loses much of its effectiveness in the push-based environment of data streams. Indeed in the continuous environment of data streams, consumption (production) of input (output) tuples follows a push-based mechanism—i.e., production (consumption) occurs without waiting for get-next requests from an embedding procedural language.

The severe limitations resulting from the problems discussed in points 1–3 above suggest that, for any SQL-based DSMS, extensions to enhance the power and flexibility of its query language are crucial to compete against the many alternatives proposed, including [1–3, 8, 9] that are discussed in Sect. 7.

Fortunately, a solution to SQL’s expressivity and flexibility problems is at hand: user-defined aggregate functions adopt a computation model based on incremental additions to their input streams, rather than the large BLOB objects of datablades. As discussed in [10, 11], User-Defined Aggregates (UDAs) can be written in an external programming language or natively in SQL itself: the native UDA definition capability makes SQL Turing-complete and NB-complete, i.e., able to express every monotonic function expressible by a Turing Machine [6]. We can finally return to the problems caused by the unbounded nature of data streams, mentioned in (b) above, and observe that windows have proved by far to be the most popular form of synopses used in DSMSs. Windows are actually not new to database languages, since SQL-2003 supports logical and physical windows on a set of built-in aggregates called OLAP functions. Many DSMSs also support these windows, and actually introduced new ones, such as slides and tumbles [1, 12], for their built-in aggregates. However, for a DSMS to address both problems (a) and (b) above effectively, its language should support these windows on arbitrary UDAs besides on built-in aggregates. Indeed, the Stream Mill system developed at UCLA supports (i) windows on arbitrary UDAs and (ii) the native definition of these UDAs in SQL [13]. Thus Stream Mill is unique in this respect, and its uniqueness is reflected in the much broader range of applications it can support efficiently: for instance, its Expressive Stream Language (ESL) can express data mining queries, sequence queries, approximate queries, etc., that are beyond the reach of other DSMSs. Examples of these advanced applications will be discussed later in this chapter, which is organized as follows.

In Sect. 2, we introduce the main extensions to SQL:2003 supported in ESL, including UDAs. In Sect. 3, we extend ESL to support different kinds of windows on arbitrary UDAs. Then, in Sects. 4 and 5, respectively, we demonstrate the effectiveness of these extensions in expressing approximate computations and advanced data mining algorithms. In Sect. 6, we describe the architecture of the Stream Mill system that supports these advanced query constructs very efficiently. Section 7 contains a broad description of related work, and it is then followed by the conclusion.

2 ESL: An Expressive Stream Language Based on SQL

ESL supports ad-hoc SQL queries and updates on database tables and continuous queries on data streams. Each data stream is declared by a CREATE STREAM declaration that also specifies the external wrapper from which data is imported and the timestamp associated with the stream. For instance, in Example 1, the data stream **OpenAuction** is declared as having **start_time** as its external timestamp.

Example 1 Declaring Streams in ESL

```
CREATE STREAM OpenAuction(
    itemID INT, sellerID CHAR(10),
    start_price REAL, start_time TIMESTAMP)
    ORDER BY start_time SOURCE ... /* Wrapper ID */;
```

In ESL, new streams can be defined from existing streams in ways similar to defining virtual views in SQL. For instance, to derive a stream consisting of the auctions where the asking price is above 1000, we can write:

Example 2 Performing Selection Operations on Streams

```
CREATE STREAM expensiveItems AS
SELECT itemID, sellerID, start_price, start_time
FROM OpenAuction WHERE start_price > 1000
```

In terms of semantics, these ESL operators produce the same results as if instead of being applied to data streams, they were applied to database tables to which new tuples are being continuously appended. Additional operators supported by ESL on data streams are (i) aggregates (built-in or user-defined) (ii) joins of a stream with a database table, and (iii) union of two or more data streams. All the operators considered so far operate on a single data stream, except for union which always returns tuples sorted by timestamp (thus its equivalent SQL statement is really UNION ALL followed by ORDER BY TIMESTAMP).

In [6], it was proven that constructs mentioned above make ESL NB-complete, thus capable of expressing all nonblocking computations on data streams. In the rest of the chapter, we therefore concentrate on these constructs, which are the most distinctive feature of ESL, and their effective use in expressing complex data stream applications. Space limitations prevent us from covering here other constructs, such as window joins, inasmuch as these constructs are less critical in terms of expressive power, and their treatment by ESL is similar to that of other DSMSs [14, 15].

2.1 User-Defined Aggregates (UDAs)

In recent years, some of the most successful SQL extensions involve aggregates, including (i) data cubes, and (ii) OLAP functions where continuous aggregates are

incrementally computed on logical and physical windows [4]. Logical and physical windows for aggregates are also provided by many DSMSs, some of which also support additional window constructs based on the notions of slides and tumbles [1, 12]. Many current DBMSs and DSMSs also allow the importation of new UDAs defined in external programming languages; however they do not support windows on such UDAs. Now, ESL brings two important improvements to the state-of-the-art by supporting (i) windows on arbitrary UDAs—including logical windows, physical windows, tumbles, and slides—and (ii) the definition of UDAs in SQL (besides external PLs).

Example 3 declares a UDA, `myavg`, equivalent to the standard `avg` aggregate in SQL, using a definition consisting of the three statement groups that are labelled `INITIALIZE`, `ITERATE`, and `TERMINATE`. Commercial DBMSs [9] and DSMSs supporting UDAs [1, 15] allow a programmer to use external procedural languages to specify the computations to be performed in `INITIALIZE`, `ITERATE`, and `TERMINATE`; this capability is also available in ESL, which however also supports native UDA definition.

In Example 3, the first line in the UDA definition declares a local table, `state`, to keep the sum and count of the values processed so far. Then, the `INITIALIZE` statement inserts the value taken from the input stream and sets the count to 1. The `ITERATE` statement updates the table by adding the new input value to the sum and 1 to the count. The `TERMINATE` statement returns the ratio between the sum and the count as the final result of computation using the `INSERT INTO RETURN` statement. `Myavg` and similar UDAs can then be used as standard SQL aggregates, with optional `GROUP BY` clause.

Example 3 Defining the standard aggregate average

```
AGGREGATE myavg(Next Int) : Real
{
    TABLE state(tsum Int, cnt Int);
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
        UPDATE state
        SET tsum=tsum+Next, cnt=cnt+1;
    }
    TERMINATE : {
        INSERT INTO RETURN
        SELECT tsum/cnt FROM state;
    }
}
```

Natively defined UDAs provide an extensibility mechanism of great power and flexibility; in fact, SQL with natively defined UDAs becomes Turing complete, and thus can express all computable queries on database tables [6]. Aggregates such as

myavg, however, are blocking and thus do not add to the expressive power of SQL in data stream applications, which instead require nonblocking aggregates.

Basically, there are two ways to turn a UDA such as **myavg** into a nonblocking UDA. The first is to modify its definition so it becomes a continuous aggregate that returns values during, rather than at the end of, the computation.

Example 4 The continuous average: a nonblocking UDA

```
AGGREGATE online_avg(Next Int) : Real
{   TABLE state(tsum Int, cnt Int);
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE: {
        UPDATE state
        SET tsum=tsum+Next, cnt=cnt+1;
        INSERT INTO RETURN
        SELECT tsum/cnt FROM state
        WHERE cnt % 200 = 0;
    }
    TERMINATE : { }
}
```

For instance, Example 4, shows a continuous version of average where results are returned every 200 tuples rather than at the end. Observe that in this definition the TERMINATE state is empty which assures that the aggregate is nonblocking and monotonic.²

The second way to deal with a blocking aggregate consists of keeping the original definition unchanged and using it to continuously recompute the aggregate on a sliding window—in a fashion similar to that of SQL:2003 OLAP functions. We next illustrate an interesting application of the first class of nonblocking aggregates; window-based applications will be discussed in the next section.

2.2 Pattern Queries

Since UDAs process tuples one-at-a-time, they are effective on physically-ordered sequences and can search for patterns in a sequence effectively. Say, for instance, that we want to find the situation where users, immediately after placing an order, ask for a rebate and then cancel the order. Finding this pattern in SQL requires two self-joins on the incoming event-stream of user activities. In general, recognizing the

²Updates and other nonmonotonic constructs can still be used freely on other database tables, such as **state**. But any operator applied to the data stream must be monotonic.

pattern of n events would require $n - 1$ joins and queries involving the joins of many streams can be complex to express in SQL. Furthermore, such queries would be very inefficient on data streams. In particular, the condition that a tuple must *immediately follow* another tuple is complex and inefficient with basic SQL but easy with UDA. For instance, consider an incoming stream of purchase actions:

```
webevents(CustomerID, ItemID, Event, Amount, Time)
```

We want to detect the pattern of an order, followed by a rebate, and immediately after that, a cancellation of the same item. Then the following nonblocking UDA can be used to return the string ‘pattern123’ with the CustomerID whose events have just matched the pattern (the aggregate will be called with the group-by clause on **CustomerID, ItemID**). This UDA models a finite state machine, where 0 denotes the failure state that is set whenever the right combination of current-state and input is not observed. Otherwise, the state is first set to 1 and then advanced step-by-step to 3, where ‘pattern123’ is returned and the computation continues to search for the next pattern.

Example 5 First the order, then the rebate, and finally the cancellation

```
AGGREGATE pattern(CustomerID Char, Next Char) : (Char, Char)
{   TABLE state(sno Int);
    INITIALIZE : {
        INSERT INTO state VALUES(0);
        UPDATE state SET sno = 1 WHERE Next= ‘order’;
    }
    ITERATE: {
        UPDATE state SET sno = 0
        WHERE NOT (sno = 1 AND Next = ‘rebate’)
            AND NOT (sno = 2 AND Next = ‘cancel’)
            AND Next <> ‘order’
        UPDATE state SET sno = 1 WHERE Next= ‘order’;
        UPDATE state SET sno = sno+1
        WHERE (sno = 1 AND Next = ‘rebate’)
            OR (sno = 2 AND Next = ‘cancel’)
        INSERT INTO RETURN
            SELECT CustomerID,‘pattern123’ FROM state WHERE sno = 3;
    }
}
```

While UDA can be effectively used to search for simple patterns, this approach can easily become prohibitive for more involved patterns.

Fortunately, powerful languages based on Kleene-* constructs were recently to facilitate the expression of complex sequential patterns. The first such language was SQL-TS [16, 17], for which powerful query optimization techniques were also developed [16, 17]. This led to an industrial SQL-change proposal [18] which has been prototyped more recently [19]. Recently proposed query languages based on Kleene-* constructs include SASE [20], SASE+ [21], Cayuga [22], CEDR [23], and

finally, K*SQL [24, 25]. Stream Mill supports K*SQL, which is the most powerful of these languages, inasmuch as it can express complex queries over relational data, and complex XML streams. K*SQL uses nested word automata³ which in turn are implemented as UDAs. For instance, the following K*SQL query could be implemented by calling UDA in Fig. 5 with a PARTITION BY **CustomerId** (where ORDER BY **Time** is implicitly assumed for our timestamp-ordered data streams).

Example 6 (Example 5 expressed in K*SQL)

```
SELECT 'modified-pattern123', X.CustomerId
FROM webevents
PARTITION BY CustomerId
AS PATTERN (X Y Z)
WHERE
  X.Event = 'order' AND
  Y.Event = 'rebate' AND Y.ItemID = X.ItemID AND
  Z.Event = 'cancel' AND Z.ItemID = Y.ItemID
```

Thus we have a simple pattern specifying the sequence of 3 events where the second (third) event immediately follows the first (second) in the separate CustomerId substream. Languages such as SASE+ [21] adopt a semantics where the next event in the pattern is simply required to follow, rather than immediately follow, the previous event. This more relaxed pattern can be expressed in K*SQL by simply changing the pattern in Example 6 to AS PATTERN (X V* Y W* Z). Here V* and W* each can match zero or more successive events,⁴ thus a clause such as Z.Time-Y.Time ≤ 60 might be advisable to limit the overall times elapses to 60 minutes. By adding F. in the WHERE clause, we can express local conditions, i.e., F.Z.ItemID = F.Y.ItemID will only require that in each occurrence of F the ItemID is the same, while different occurrences of F can have different values of ItemID.

This seemingly simple extension, makes K*SQL strictly more expressive than its counterparts. Fortunately, despite its higher expressive power, K*SQL has also proved to be highly amenable to efficient execution over high-volumes of stored sequences and data streams [25]. For these reasons, and due to the appealing syntax of K*SQL for sequence queries, the Stream Mill system also supports special built-in UDAs that can efficiently execute K*SQL queries [24]. Likewise, it was previously shown that queries expressed in SQL-TS [16] could be mapped into equivalent ESL queries through the use of specialized UDAs [28]. Similarly, UDAs were used in [28] to implement the FSA computation used by Yfilter to support multiple queries on streaming XML data, thus unifying the processing of these two kinds of streams.

Thus, UDAs have proved to be a key extension of SQL. Furthermore, ESL greatly enhances their power and versatility for data stream applications by providing the flexible window mechanisms for arbitrary UDAs that are discussed next.

³Nested words [26] and visibly-pushdown automata [27] can model data with both sequential and hierarchical structures, such as XML, RNA sequences or procedural software traces.

⁴However, expressing “immediately following” in SASE+ is significantly more difficult.

3 Window Aggregates and Their Applications

Following SQL:2003, ESL uses the OVER clause to specify (i) the type of window (i.e., logical or physical), (ii) the size of the window (using a time span for logical windows or the number of tuples for physical ones), and (iii) the columns in the partition-by clause (if any). However, for data streams, the ORDER BY clause can be omitted, since data streams are always ordered by their timestamps.

In Example 7, we have a physical window of 100 items, consisting of the current tuple and the 99 rows preceding it; a separate window is maintained for each seller, as specified by the (PARTITION BY **sellerID**) clause.

Example 7 For each seller, maintain the max selling price over the last 100 items sold.

```
CREATE STREAM LastTenAvg
SELECT sellerID, max(price) OVER
    (PARTITION BY sellerID ROWS 99 PRECEDING)
FROM ClosedPrice;
```

By replacing, say, ‘ROWS 99’ with ‘RANGE 5 MINUTES’, we would instead specify a logical window of five minutes.

Because of their many uses, the notions of window slides and tumbles [1, 12], are now supported in many DSMSs, although they go beyond the SQL:2003 standards. These constructs are supported in ESL for arbitrary UDAFs, using a ‘SLIDE’ declaration in the window clause. For instance, the following example is similar to the previous one in every aspect, but the fact that results are now returned every 10 rows, rather than after each row as in the previous case.

```
CREATE STREAM LastTenAvg
SELECT sellerID, max(price) OVER
    (PARTITION BY sellerID ROWS 99 PRECEDING SLIDE 10)
FROM ClosedPrice;
```

In this example, the size of the slide (10) is smaller than the overall size of the window (100). Tumbles instead occur when the size of the slide exceeds that of the window. For instance, to break the input stream into blocks of size 600 and return the average of the last 600 input tuples at the end of each block, we can specify **ROWS 599 PRECEDING SLIDE 600**. Thus the size of the window and the slide are both 600, and the input data stream is partitioned into windows of size 600 and results are returned every 600 tuples.

Window Aggregates

Providing efficient, integrated management and support for an assortment of different windows represents an interesting research problem that in the past has

been addressed only for specific built-in aggregates or specific classes of aggregates [1, 12, 29]. A naive approach to implement windows on arbitrary UDAs requires the user to write six versions of an aggregate, one for each combination of (logical|physical \times tumble|slide|no-slide). With ESL, however, users only need to write at most the following two versions: (a) the base version of the UDA, and (b) an **optional** window-optimized version. The second version (b) allows the user to perform delta-maintenance on arbitrary UDAs, which can lead to significant performance improvement. The ESL compiler utilizes these two definitions to provide integrated support for (i) general optimization tasks that are applied to all window aggregates, along with (ii) user-specified optimizations that are specified for a particular UDAs. We next illustrate how this is accomplished using the MAX aggregate as an example whose base definition is shown below:

Example 8 Base Definition for MAX

```
CREATE WINDOW AGGREGATE max (Next Real) : Real
{   TABLE current(CVal real);
    INITIALIZE : {
        INSERT INTO current VALUES (Next);
    } /* the value in is the first max */
    ITERATE : {
        UPDATE current set CVal = Next
        WHERE CVal < Next;
        INSERT INTO RETURN
        SELECT CVal FROM state;
    }
}
```

The Stream Mill system provides uniform support for physical and logical windows, thus the six combinations above degenerate to three, which we discuss next. The simplest one is that of tumbles, i.e., the case in which the UDA is called over a window of size smaller or equal to that of its slide. For instance, say that MAX is called on a window of 600 tuples and the size of its slide is $S \geq 600$. In this case, ESL will use the base definition of MAX in Example 8 and return the result after 600 tuples. Then the next $S - 600$ input tuples are ignored and the computation restarts from $(S - 600 + 1)$ th tuple and goes for another 600 tuples, and so on. Examples illustrating the use of tumbles in clustering and ensemble-based classification are given in Sects. 5.1 and 5.2.

The second case is that of Example 7, in which the UDA is called without the slide construct. A naive implementation consists in buffering all the window tuples into an inwindow table.⁵ Then, for each arriving tuple, base MAX aggregate is recomputed over the tuples in inwindow table. While this approach is correct, it is

⁵Newly arriving tuples are inserted into and expiring tuples are removed from the inwindow table automatically by the system for efficiency.

obviously inefficient. Thus ESL allows users to define a specialized version of the UDA which uses the values of tuples leaving the window to perform delta maintenance. For an aggregate such as sum, this delta maintenance involves the subtraction of the expiring value from the current sum. For MAX and more sophisticated aggregates the delta maintenance is more complex but nevertheless quite beneficial. As shown in Example 9, the delta maintenance is performed in a special state called EXPIRE. In our window version of MAX, shown in Example 9, the EXPIRE event does not require any action. The expiring tuple is simply discarded (automatically, by the system). The ITERATE state of the UDA only keeps tuples that can potentially be the maximum in the window. Thus, the oldest tuple in the inwindow table is the maximum in the current window.

The result of the aggregate is the same whether this delta computation is performed as soon as a tuple expires, later when a new tuple arrives, or anywhere in between these two instants. ESL takes advantage of this freedom to optimize execution.

Example 9 MAX with Windows

```
CREATE WINDOW AGGREGATE max (Next Real) : Real
{   TABLE inwindow(wnext real);
    INITIALIZE : {
        INSERT INTO RETURN VALUES (Next);
    } /* the system adds new tuples to inwindow */
    ITERATE : {
        DELETE FROM inwindow WHERE wnext ≤ Next;
        INSERT INTO RETURN VALUES (oldest());
    }
    EXPIRE: { } /*expired tuples are removed automatically*/
}
```

In the definition of window aggregates, EXPIRE is treated as an event that occurs once for each expired tuple—and the expired tuple is removed as soon as the EXPIRE statement completes execution. ESL also provides the built-in predicate **oldest()** which selects the oldest tuple among the tuples of **inwindow**: **oldest().wnext** delivers the **wnext** column in this tuple. If the tuple has only one column then the system allows using **oldest()**, i.e., without the column name.

Upon arrival of a new tuple, the system first proceeds at executing any outstanding EXPIRE events. The ITERATE statements are next executed on this newly arrived tuple. After the ITERATE statements, the new tuple is put into the **inwindow** buffer.

This delta-maintenance approach to window aggregates is also used in the implementation of basic ESL built-in aggregates. For instance, in case of the MAX aggregate, we eliminate tuples in the buffer that are dominated by more recent tuples—thus reducing the size of the buffer from W to $\log(W)$, where W denotes the size of the window. Frequently, the approach is also effective on more complex UDA s , such as the approximate frequent-items example discussed in Sect. 4.

For the third case, i.e., where the specified slide size is less than the window size, ESL utilizes the optimization that has been proposed in [12]. This optimization involves dividing the window in smaller panes. Window version of the UDA is used to perform the delta computation on the results of the base version of the UDA on each smaller pane [30]. Using these powerful UDAs we can also implement the ‘negative tuple’ semantics⁶ that has also been considered for windows [31].

The introduction of powerful analytics SQL:2003 have illustrate the need for DBMS to support a wider range of aggregates than those supported in SQL-2, and the important role that windows play in this context. However, windows aggregates play an even more important role in DSMS particularly those that must support complex tasks with QoS guarantees: in fact, window UDAs can support effectively (i) approximate computations, (ii) load shedding, and (iii) complex decision support and mining task. Our discussion in the rest of the chapter will focus on these topics.

4 Approximation and Sketch Aggregates

In order to assure QoS against high arrival rates and bursts in the incoming data streams, DSMS rely on (i) load shedding, and (ii) approximation techniques, which dovetail with and UDA-oriented architecture. Indeed, Stream Mill provides an architecture where load shedding is integrated with UDAs [32], supporting error models that accommodate different requirements for multiple users, different sensitivities to load shedding, and different penalty functions. By incorporating a priori statistics of data streams, the Stream Mill system can provide QoS guarantee for a large class of queries, including traditional SQL aggregates, statistical aggregates and data mining functions [32]. A Bayesian approach can be used to combine the past statical information about query answers and further boost the quality of the a priori estimations [33].

We now leave the discussion of load-shedding which would take us beyond the scope of this paper to concentrate on sketches and other approximate aggregates.

EH Sketches

Many data mining techniques require counting the frequency of different items in a window. Since computing the exact counts would require storing the whole window to determine the tuples leaving the window. For windows that are too large in size approximate counting aggregates can be used instead. In [34], Datar et al. proposed the Exponential Histogram (EH) sketch algorithm for approximating the number of 1’s in sliding windows of a 0–1 stream and showed that for a δ -approximation of the

⁶Besides returning values when new tuples arrive, under such semantics, new revised aggregate values could be produced as soon as a tuple expires out from the time-based window.

number of 1's in the current window, the algorithm needs $O(\frac{1}{\delta} \log W)$ space, where W is the window size.

The EH sketch consists of an ordered list of buckets or *boxes*. Every box in an EH sketch basically carries on two types of information, a time interval and the number of observed 1's in that interval. The intervals for different boxes do not overlap and every 1 in the current window should be counted in exactly one of the boxes. Boxes are sorted based on the start time of their intervals. For each new coming 1, EH creates a new box with size one. Then the algorithm checks if the number of boxes with the same size exceeds $k/2 + 2$ (where $k = \frac{1}{\delta}$), it merges the oldest two such boxes. The merge operation adds up the size of the boxes and merges their intervals. The final estimation of the number of 1's would be the aggregate size of boxes minus half of the oldest box's size. Note that before reporting the results, we discard the boxes which do not overlap the current window. Thus an EH UDA can be expressed by the ESL code below, and then called in a way similar to built-in SQL3 aggregates.

Example 10 Counting by Exponential Histograms

```

STREAM zeroOnes(val Int, t Timestamp);
AGGREGATE EHCount(next Int, t Timestamp, k Int):{
    WINDOW EH(h Int, t Timestamp) ORDER BY t;
    TABLE memo(last Int, total Int) MEMORY VALUES (0,0);
    AGGREGATE merge(next Int, t Timestamp, k Int):{
        /* state table stores the current box, count, and the last two timestamps */
        TABLE state(h Int, cnt Int, t1 Timestamp, t2 Timestamp) MEMORY;
        INITIALIZE:
        {INSERT INTO state VALUES(next, 1, t, NULL);}
        ITERATE:{
            UPDATE state SET cnt=cnt+1, t2=t1, t1=t WHERE h=next;
            /* should early return if true */
            UPDATE state SET h=next, cnt=1, t1=t
                WHERE h <> next AND (SELECT cnt FROM state) < k/2+2;
            /* should early return if true */
            /* if current count is k/2+2, delete the last box, and double the next-to-last one */
            DELETE FROM EH h WHERE h.t = (SELECT t1 FROM state)
                AND h <> next AND (SELECT cnt FROM state) = k/2 + 2;
            UPDATE EH h SET h=h*2 WHERE SQLCODE = 0
                AND h.t = (SELECT t2 FROM state);
            UPDATE state SET h=next, cnt=2, t1=t WHERE SQLCODE = 0; }
        }; /* the end of merge aggregate*/

        INITIALIZE:ITERATE:{

            INSERT INTO EH VALUES(next, t) WHERE next > 0; /* ignore 0's */
            UPDATE memo SET total = total + next;
            SELECT merge(h, t, k) OVER (ORDER BY t DESC) FROM EH;
            /* Update last pointer due to merge */
            UPDATE memo SET last = (SELECT max(h) FROM EH);
            INSERT INTO Return SELECT total FROM memo; }
    }
}

```

```

EXPIRE:{  

    UPDATE memo SET last = h/2  

        WHERE (SELECT count(1) FROM EH h WHERE h.h = last) =1;  

    /* update total pointer */  

    UPDATE memo SET total = total-h/2-last/2 }  

}  

/*Calling the aggregate just defined from an ESL statement */  

SELECT EHCount(val, t, 2) OVER (RANGE 10 MINUTE) FROM zeroOnes;

```

This sketch is later used in several other structures and algorithms listed in [35]. Recently, [36] has used EH to generate an approximate B-bucket equi-depth histogram for data streams with sliding windows. The proposed approach which is called BAr-Splitting Histograms (BASH) is based on dividing the acceptable input range into several chunks or *bars* in a way that each bar contains roughly equal number of items. The number of bars are limited to fix value which is greater than B for improving the accuracy. To keep the size of bars roughly the same as the stream passes, a splitting/merging technique is employed to split big bars, and merge adjacent small bars. BASH provides a very fast and space-efficient equi-depth histogram particularly for high speed data streams.

Approximate Frequent Items

The problem of determining the frequent items in a data stream is important in many applications and several algorithms have been proposed to deal with the common situation where there is enough memory for the frequent items but not for all items [37–39]. Here we focus primarily on [37], which is a windowed approximate frequent items algorithm suitable for delta computation. The algorithm, shown with ESL code in Example 11, maintains k hash-tables over the current window. Each hash-table has a corresponding hash-function. Each hash entry in the hash-tables is an integer, which is used as a counter. When an item enters the window, we iterate through the k hash-functions and determine the k key values. For each key value, we increment the counter at that location in the corresponding hash-table. Similarly, when an item expires out of the window, we decrement the corresponding k counters. Finally, the approximate frequency of an item is determined by taking the minimum value of the k counters. Note, that this minimum value may over estimate the frequency of the item, if all k keys have at least one other item mapped to it. This algorithm can also be viewed as a bloom-filter with two exceptions: (i) there are k different hash-tables instead of just one and (ii) each entry is an integer(counter) as opposed to a bit. In addition to the delta maintenance property, the algorithm provides bounded error estimates. Thus, given the available amount of memory we can estimate the expected error.

Example 11 Approximate Frequency Count

```

STREAM items(item Int); /* Stream of items */
TABLE hash_tables(index1 Int, index2 Int, cnt Int) MEMORY;
/* the k hash tables index2 goes from 1 to k*/
TABLE hs(h Int, ah Int, bh Int) MEMORY; /* constants for hash functions */
/* table initialization omitted */

/* Windowed aggregate that maintains the hash-tables */
WINDOW AGGREGATE MaintainHashes(k Int):Int {
    /* an aggregate that updates a certain hash entry */
    AGGREGATE updateCnt(k Int, h Int, ah Int, bh Int, val Int):Int {
        INITIALIZE: ITERATE: {
            UPDATE hash_tables SET cnt = cnt+val
            WHERE index1 = ((ah*k+bh)%31)%4 AND index2 = h }
        };
        INITIALIZE: ITERATE: { /* new item entering the window */
            SELECT updateCnt(k, h, ah, bh, 1) FROM hs
        }
        EXPIRE: { /* item expiring */
            SELECT updateCnt(k, h, ah, bh, -1) FROM hs
        };
        /* Calling the UDA just defined*/
        SELECT MaintainHashes(item) OVER (ROWS 29 PRECEDING)
        FROM items;
    };
}

```

5 Mining Data Streams

Data stream mining represents an important area of current research, and the topic of many recent papers, which primarily focus on devising mining algorithms that are fast and light enough to be executed continuously and produce real-time or quasi real-time responses. However, online data mining represents such a difficult issue for DSMS that no system before Stream Mill [40] has claimed success in this important application. Many of the problems facing DSMS are similar to those of DBMSs that in mid-1990s were unable to extend the success of SQL on OLAP applications to data mining applications. Indeed performing data mining tasks through the DBMS-supported constructs and functions was exceedingly difficult [41], whereby in a visionary 1996 paper [42], Imielinski and Mannila called a major research effort to produce quantum leap in the functionality and usability of DBMSs, whereby mining queries can be supported with the same ease of use as other relational queries are now supported. The notion of “Inductive DBMS” was thus born, which inspired much research [43], while vendors have been working on providing some data mining functionality as part of their DBMS [44].

The Stream Mill DSMS supports a powerful data stream mining workbench called SMM which is open and extensible. The first ingredient of SMM is a library of powerful data stream mining methods defined as window UDA. New methods can be defined using ESL, or procedural languages; in either case the definition follows the standard INITIALIZE, ITERATE, TERMINATE, and EXPIRE templates of aggregates previously described. For the analysts and other users who want to work at higher level of abstraction, SMM support mining models [40, 45]. Mining methods and models are discussed next.

5.1 Density-Based Clustering (DBScan)

DBScan represents a popular clustering algorithm that can be successfully applied to mining and monitoring data streams [46]. Let us assume we have a stream of two-dimensional data, where more than **minPts** points occurring in close proximity (i.e., at distance less than **eps**) of each other are assigned to the same cluster, while sparse points are instead classified as outliers. To monitor changes in the incoming stream of two-dimensional data, we employ DBScan algorithm as follows: (i) partition the stream into blocks containing the same number of tuples, (ii) cluster the data in each block, and (iii) monitor the appearance/disappearance of new/old clusters and changes in cluster population between successive blocks. The first two tasks are accomplished by the following ESL statement that invokes the **dbscan** aggregate on input data stream **Stream_of_Points(Xvalue, Yvalue, TimeStamp)**:

```
/*call dbscan with minPts = 10 and eps = 50 */
SELECT dbscan(Xvalue, Yvalue, 0, 10, 50)
    OVER(ROWS 999 PRECEDING SLIDE 1000 )
FROM Stream_of_Points
```

Here 10 and 50 are the example values we assign to two important parameters for the DBScan Algorithm, **minPts** and **eps**, respectively. The third argument is for book-keeping purposes. Observe that since the size of the slide is the same as that of the window, this is a tumble. Therefore the Stream Mill system will use the base definition of DBScan, shown below, independent of whether a window version is available or not.

Given the two parameters **eps** and **minPts**, the DBScan algorithm works as follows: pick an arbitrary point **p** and find its neighbors (points that are less than **eps** distance away). If **p** has more than **minPts** neighbors then form a cluster and call DBScan on all its neighbors recursively. If **p** does not have more than **minPts** neighbors then move to other un-clustered points in the database. Note, this can be viewed as a depth-first search.

```
AGGREGATE dbscan(iX Real, iY Real, Flag Int, minPt Int, eps Int): Int
{
    TABLE closepts(X2 real, Y2 real, C2 Int) MEMORY;
    INITIALIZE: ITERATE: {
        /* Find neighbors of the given point */
```

```

INSERT INTO CLOSEPNTS SELECT X1, Y1, C1 FROM points
    WHERE sqrt((X1-iX)*(X1-iX) + (Y1-iY)*(Y1-iY)) < eps;
/* If there are more than minPt neighbors, form a cluster */
UPDATE clusterno SET Cno= Cno+1 /* new cluster number*/
    WHERE Flag=0 AND SQLCODE=0 /* A new cluster */
        AND minPt < (SELECT count(C2) FROM closepnts);
/* Assign these neighboring points to this cluster */
UPDATE points SET C1 = (SELECT Cno FROM clusterno)
    WHERE points.C1=0 AND
        EXISTS (SELECT S.X1 FROM closepnts AS S
            WHERE points.X1=S.X2 AND points.Y1=S.Y2 )
        AND minPt < (SELECT count(C2) FROM closepnts);

/* Call dbscan recursively */
SELECT dbscan(X2, Y2, 1, minPt, eps)
    FROM closepnts, points
    WHERE X1 = X2 AND Y1=Y2;
DELETE FROM closepnts;
}
};

/*end dbscan*/

```

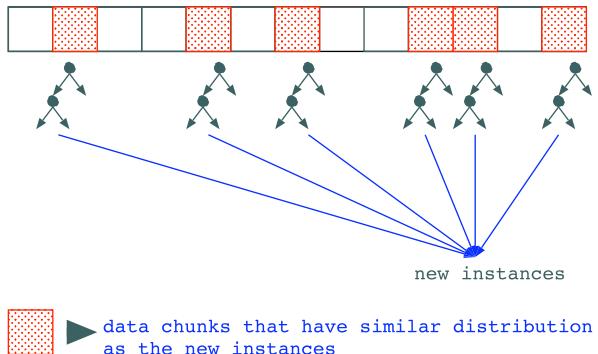
5.2 Mining Data Streams with Concept Drift

Since streaming data is characterized by time-changing concepts, a basic challenge faced by data mining algorithms is to model and capture the time-evolving trends and patterns in the streams, and make time-critical predictions. One approach is to incrementally maintain a model for the time-changing data. In this case the model is learned from data in the most recent window. This approach has several weak points. First, given that data arrives at a high speed, incremental model maintenance is usually a costly task, especially for learning methods such as the decision tree algorithm, which are known to be unstable. Second, models trained from the data in a window may not be optimal. If the window is too large, it may contain concept drifts; if it is too small, it may result in over-fitting.

A more effective approach consists in using an ensemble based model whereby we partition the stream into fixed size data chunks and learn a model from each chunk. We combine models learned from data chunks, whose class distribution is similar to the most recent training data, to be our stream classifier (as shown in Fig. 1). This approach reduces classification error in the concept-drifting environment. We use ESL to implement this approach [47] effectively, which takes full advantage of off-the-shelf classifier packages and other procedural routines.

The seemingly complex solution described above can be implemented in ESL in a very succinct way. We assume each data record in the stream is in the form of $(\mathbf{a}_1, \dots, \mathbf{a}_n, \mathbf{L})$, where $\mathbf{a}_1, \dots, \mathbf{a}_n$ are attribute values, and \mathbf{L} is the class label. If $\mathbf{L} =$

Fig. 1 Mining streams with concept-drift



TBA then it is a testing example, otherwise it is a training example. In Example 12, we express the algorithm in one SQL statement.

Here, we call UDA **ClassifyStream** with keyword **SLIDE**, which implements data partitioning on the stream, via tumbles of size 1000. We assume classifiers, together with their weights, are stored in a table called **ensemble**. In UDA **classifystream**, we use classifiers in the **ensemble** whose weights are above a given threshold to classify each test example, where **Classify** is a UDA for classifying static data [11]. Once we reach the end of a data partition, we learn a new classifier from the training data in the partition, and we reset the weight of each classifier in the **ensemble** proportional to its accuracy in classifying the most recent training data. The freshly weighted classifiers will then be used to classify data in the next partition.

Example 12 A Terse Expression for Complex Classifier Ensembles

```
SELECT ClassifyStream(S.*)
    OVER (ROWS 999 PRECEDING SLIDE 1000)
FROM stream AS S;
```

Example 13 UDA **classifystream**

```
AGGREGATE ClassifyStream(a1, ..., an, L) : Int
{   TABLE temp(a1, ..., an, L);
    INITIALIZE : ITERATE : {
        INSERT INTO RETURN
        SELECT sum(E.Classify(a1, ..., an) × E.weight) /
            sum(E.weight)
        FROM ensemble AS E
        WHERE L = TBA AND E.weight ≥ threshold;
        INSERT INTO TEMP VALUES (a1, ..., an, L);
    }
    TERMINATE : {
        INSERT INTO ensemble
```

```

SELECT learn(T.*)
  FROM TEMP AS T
 WHERE T.L <> TBA;
UPDATE ensemble AS E SET E.weight =
  (SELECT 1-avg(|E.Classify(T.*)-T.L|)
   FROM TEMP AS T
  WHERE T.L<>TBA);
}
}

```

5.3 Mining Models

The integration of mining methods into SMM is made simple via the Mining Model Definition Language which support the declaration of mining models [40]. Each mining model instance defines (i) which mining UDA will be used in the task, (ii) the parameter values and ancillary information they will use, and (iii) the flow of stream data between these methods [40, 45].

Seldom *flows* need to be specified for complicated mining tasks. Consider a more advanced mining method such as an ensemble based weighted bagging (EBWB) [47], which is supported by SMM to improve the accuracy of classifiers in the presence of concept drifts and shifts. With EBWB, instead of maintaining a single classifier, the user maintains several small classifiers, whose classification is combined later using some kind of weighted voting. This approach assures a better adaptation in the presence of concept-shift and concept-drift, since new classifiers can be continuously trained based on the latest statistics, while older or inaccurate classifiers can be retired. Note that specifying the various steps required for weighted bagging represents a daunting task for analysts and less experienced users. Therefore, MMDL supports specification of one or more mining *flows* within the mining model definition. These complex mining processes only have to be specified once during model definition and can be reused by all users. *Flows* have been essential in definition of many built-in mining methods, such as SWIM for association rule mining [48], in SMM [40]. At the best of our knowledge SMM's ability to support data stream mining algorithms is unique among DSMS and CEP systems. On the other hand, systems such as MOA [49] provide a flexible and user-friendly environment for evaluating algorithms for data stream mining and for the incremental mining of data sets; however such systems are not DSMS designed to support QoS for continuous queries over extended periods of time.

6 The Stream Mill System

The architecture of the Stream Mill system consists of a single server and multiple clients.

The Client

Users interact with the server through the query editor—marked as α in Fig. 2. The query editor allows the user to perform the following tasks: logging in and out of the system, defining streams, queries, aggregates, starting and stopping queries, etc. Results of these tasks are shown in the query editor’s status pane, by default. The client also provides a set of GUI modules to display the workflow and the results of the continuous queries in a graphical form, e.g., β in Fig. 2. Several performance meters (marked as γ) are also at hand to continuously monitor traffic, memory utilization, queue length, and related measures of server performance.

The Server

The bulk of Stream Mill R&D efforts focused on the Server, which supports the following functional modules:

Query Compiler/Optimizer

The compiler is responsible for parsing and compiling continuous queries and generating/modifying the query graph that describes how continuous queries are implemented by operators that take tuples from their input buffers and push them into their output buffers (in cooperation with the Buffer Manager). These operators are implemented as C/C++ functions compiled into dynamic libraries, which are then invoked by the Execution Scheduler. After careful optimizations, natively defined UDA_s on the average execute nearly as well (a 30 % slowdown) as UDA_s externally defined in C++ and better than those defined in Java

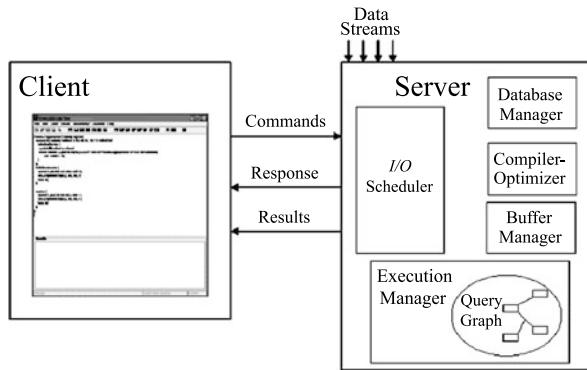
Buffer Manager

The Buffer Manager is responsible for managing the stream tuples as in-memory queues. Most tuples are processed and removed from these buffers as quickly as possible to free space and to reduce latency. However, when windows are used in the query, tuples must be retained main memory for extended period of time. This is the task of the *Window Manager*, which is also responsible for supporting delta-maintenance constructs for these windows, window sharing, and suitable paging policy.

Execution Scheduler

The Execution Scheduler is responsible for deciding which operator from the query graph executes next and how many input tuples it will consume. This decision reflects the optimization criterion selected, which in turn reflects priorities specified

Fig. 2 The stream mill system architecture



by the user and also the load conditions currently experienced in the system. For instance, a low-memory condition might force the scheduler to change from a scheduling policy that minimizes response time to one that minimizes memory [50]. This change might require a modification (e.g., partitioning) of the query graph; Stream Mill is capable of performing this adjustment very quickly, efficiently, and without stopping the execution of the continuous queries [51]. Stream Mill's flexible execution model also supports on demand-generation of timestamp to minimize idle-waiting in operators such as union and joins [51].

Other important modules which are part of the system include the *I/O Scheduler*, which is responsible for managing incoming and departing data streams, and the interaction between the server and the outside world, including the Stream Mill Client. The *Database Manager*, based on *ATLAS* and *Berkeley DB*, is used to support database queries and spanning applications. Various extensions and improvements being added include new data stream mining algorithms, load shedding extensions, and better GUI primitives.

7 Related Work

Data Stream Management Systems (DSMSs) represent a vibrant area of current research comprising several subareas. Because of space limitations and the availability of authoritative surveys [5, 52], we will here focus on previous works that are most relevant to ESL.

The Tapestry project [53, 54] that was the first to discuss ‘queries that run continuously over a growing database’ with append-only relations as the basic data model for data streams. The append-only data model was then adopted in many projects, including Tribeca [55], and Telegraph [56]. Likewise, OpenCQ [57] and Niagara Systems [58] are designed to support continuous queries for monitoring web sites, and the Chronicle data model uses append-only ordered sets of tuples (chronicles) [59].

With respect to languages for continuous queries, the use of SQL and its dialects is predominant not only for DSMS of relational DBMS lineage, but also for the

various systems of mongrel lineages known as CEP systems.⁷ In reality, however, the apparent popularity of SQL is restricted by many exceptions and limitations. For instance, extensions to C++ are used in Hancock [2], while systems focusing on streaming XML data use XQuery or Xpath [8, 60]. Finally, CEP systems tend to support SQL only as a tool of convenience for simple applications, while they rely on some Java-based language for more serious applications and system extensions. Even in DSMS with a relational database lineage, we find variations and limitations. For instance, Tribeca relies on operators adapted from relational algebra [55], while active database rules are used in OpenCQ [57]. Furthermore, the influential Aurora/Borealis project [61] focuses on providing an attractive graphical interface to define a network of continuous query operators, where the user can then request an equivalent SQL program to be produced from this.

Another influential DSMS project is STREAM; this system and its Continuous Query Language (CQL) [14] features several syntactic variations from the SQL:2003 standards, and from the append-only model, by proposing an approach based on database queries over continuously sliding windows.

Unlike many other DSMS projects, however, the Stream Mill project seeks to preserve the syntax and semantics SQL standards as far as possible. In this respect, our project is similar to the very influential Gigascope [15] project, which has also adopted the append-only model as ESL does. But unlike Gigascope, which was designed primarily for network analysis and management, ESL strives to serve a much wider range of applications, including applications not supported by other DSMSs, such as pattern queries on relational and XML streams [25, 28], and data mining queries [40]. Thus, while Gigascope relies on SQL-2 style of aggregates, ESL adopts the SQL:2003 constructs for windowed aggregates; the same constructs are then applied to UDA producing a compact language that is Turing complete on stored data and NB-complete on streaming data [6, 62].

8 Conclusion

A key contribution of ESL and Stream Mill is proving that rather limited extensions enable database query languages to support effectively a very wide range of data stream applications, by providing levels of expressive power and generality that match or surpass those of other query languages and systems proposed for data stream and publish/subscribe applications [1–3, 8, 9]. The merits of ESL extensions are supported by theoretical results [6, 62] and demonstrated by important applications that are beyond the reach of other DSMS, including continuous data mining queries [40], sequence queries on the nested-word and the XML model [24, 25, 28], and algorithms for synopsis maintenance [36]. This significant leap in power and generality has been achieved while preserving the basic append-only-table semantics for data streams and minimizing extensions w.r.t. SQL:2003 standards—as

⁷http://en.wikipedia.org/wiki/Complex_event_processing.

needed to facilitate the writing of applications that span both databases and data streams.

The Stream Mill prototype is now fully operational and supports (i) continuous queries on data streams [63], (ii) ad hoc queries on database tables, and (iii) ad hoc queries on table-like concrete views defined on data streams. More information on (iii), time-series queries, and XQuery on SAX in Stream Mill is available from the project web site [13].

Acknowledgements Thanks are due to Yijian Bai, Yanpei Law, Stefano Emiliozzi, Shu Man Li, Vincenzo Russo, and Xin Zhou for their many contributions to the system and its enabling technology.

References

1. D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S.Z. Aurora, A new model and architecture for data stream management. *VLDB J.* **12**(2), 120–139 (2003)
2. C. Cortes, K. Fisher, D. Pregibon, A. Rogers, Hancock: a language for extracting signatures from data streams, in *SIGKDD* (2000), pp. 9–17
3. P. Felber, P. Eugster, R. Guerraoui, A. Kermarrec, The many faces of publish/subscribe. *ACM Comput. Surv.* **35**(2), 114–131 (2003)
4. ISO/IEC. Database languages—SQL, ISO/IEC 9075-*:2003 (2003)
5. B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in *PODS* (2002), pp. 1–16
6. Y.-N. Law, H. Wang, C. Zaniolo, Data models and query language for data streams, in *VLDB* (2004), pp. 492–503
7. I. Information Technologies, Illustra user’s guide, in *1111 Broadway, Suite 2000*, Oakland, CA (1994)
8. D. Florescu, C. Hillery, D. Kossmann et al., The BEA/XQRL streaming xquery processor. *VLDB J.* **13**(3), 294–315 (2004)
9. Oracle. Oracle9i application developer’s guide advanced queuing. Oracle, Redwood Shores, CA, USA (2002)
10. H. Wang, C. Zaniolo, Using SQL to build new aggregates and extenders for object-relational systems, in *VLDB* (2000), pp. 166–175
11. H. Wang, C. Zaniolo, Atlas: a native extension of sql for data mining, in *Proceedings of Third SIAM Int. Conference on Data Mining* (2003), pp. 130–141
12. J. Li, D. Maier, K. Tufte, V. Papadimos, P.A. Tucker, Semantics and evaluation techniques for window aggregates in data streams, in *SIGMOD Conference* (2005), pp. 311–322
13. Stream mill home. <http://wis.cs.ucla.edu/stream-mill>
14. A. Arasu, S. Babu, J. Widom, Cql: a language for continuous queries over streams and relations, in *DBPL* (2003), pp. 1–19
15. C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, O. Spatscheck, Gigascope: high performance network monitoring with an sql interface, in *SIGMOD* (ACM, New York, 2002), p. 623
16. R. Sadri, C. Zaniolo, A. Zarkesh, J. Adibi, Optimization of sequence queries in database systems, in *PODS* (2001)
17. R. Sadri, C. Zaniolo, A.M. Zarkesh, J. Adibi, Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.* **29**(2), 282–318 (2004)
18. F. Zemke, A. Witkowski, M. Cherniak, L. Colby, Pattern matching in sequences of rows, in *Sql Change Proposal* (2007). <http://www.sqlsnippets.com/en/topic-12162.html>

19. N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, N. Tatbul, Dejavu: declarative pattern matching over live and archived streams of events, in *SIGMOD Conference* (2009), pp. 1023–1026
20. E. Wu, Y. Diao, S. Rizvi, High-performance complex event processing over streams, in *SIGMOD Conference* (2006), pp. 407–418
21. D. Gyllstrom, J. Agrawal, Y. Diao, N. Immerman, On supporting kleene closure over event streams, in *ICDE* (2008), pp. 1391–1393
22. A.J. Demers et al., Cayuga: a high-performance event processing engine, in *SIGMOD Conference* (2007), pp. 1100–1102
23. R.S. Barga et al., Consistent streaming through time: a vision for event stream processing, in *CIDR* (2007), pp. 363–374
24. B. Mozafari, K. Zeng, C. Zaniolo, K^{*}SQL: a unifying engine for sequence patterns and XML, in *SIGMOD Conference—Demo Track* (2010), pp. 1143–1146
25. B. Mozafari, K. Zeng, C. Zaniolo, From regular expressions to nested words: unifying languages and query execution for relational and XML sequences. *Proc. VLDB Endow.* **3**(1), 150–161 (2010)
26. R. Alur, P. Madhusudan, Adding nesting structure to words, in *Developments in Language Theory* (2006)
27. R. Alur, P. Madhusudan, Visibly pushdown languages, in *STOC* (2004), pp. 202–211
28. X. Zhou, H. Thakkar, C. Zaniolo, Unifying the processing of XML streams and relational data streams, in *ICDE* (2006), p. 50
29. U. Srivastava, J. Widom, Memory-limited execution of windowed stream joins, in *VLDB* (2004), pp. 324–335
30. Y. Bai, H. Thakkar, C. Luo, H. Wang, C. Zaniolo, A data stream language and system designed for power and flexibility, in *CIKM* (2006), pp. 337–346
31. L. Golab, M. Tamer Özsu, Update-pattern-aware modeling and processing of continuous queries, in *ACM SIGMOD Conference* (2005), pp. 658–669
32. B. Mozafari, C. Zaniolo, Optimal load shedding with aggregates and mining queries, in *ICDE* (2010), pp. 76–88
33. Y.-N. Law, C. Zaniolo, Improving the accuracy of continuous aggregates and mining queries on data streams under load shedding. *Int. J. Bus. Intell. Data Min.* **3**(1), 99–117 (2008)
34. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows: (extended abstract), in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2002), pp. 635–644
35. C. Aggarwal, *Data Streams: Models and Algorithms* (Springer, Berlin, 2007)
36. H. Mousavi, C. Zaniolo, Fast and accurate computation of equi-depth histograms over data streams, in *EDBT* (2011), pp. 69–80
37. C. Jin, W. Qian, C. Sha, J.X. Yu, A. Zhou, Dynamically maintaining frequent items over a data stream, in *Proceedings of the 12th ACM Conference on Information and Knowledge Management (CIKM)* (2003)
38. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in *International Colloquium on Automata, Languages, and Programming (ICALP)* (2000), pp. 508–515
39. G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, in *PODS* (2003), pp. 296–306
40. H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, S.M.M. Carlo Zaniolo, A data stream management system for knowledge discovery, in *ICDE* (2011), pp. 757–768
41. S. Sarawagi, S. Thomas, R. Agrawal, Integrating association rule mining with relational database systems: alternatives and implications, in *SIGMOD* (1998)
42. T. Imielinski, H. Mannila, A database perspective on knowledge discovery. *Commun. ACM* **39**(11), 58–64 (1996)
43. C. Zaniolo, Mining databases and data streams with query languages and rules—invited paper, in *KDID 2005: Knowledge Discovery in Inductive Databases, 4th International Workshop*. Lecture Notes in Computer Science, vol. 3933 (Springer, Berlin, 2006), pp. 24–37
44. Z. Tang, J. MacLennan, P.P. Kim, Building data mining solutions with OLE DB for DM and XML for analysis. *SIGMOD Rec.* **34**(2), 80–85 (2005)

45. H. Thakkar, B. Mozafari, C. Zaniolo, Designing an inductive data stream management system: the stream Mill experience, in *SSPS* (2008), pp. 79–88
46. H.-P. Kriegel, M. Ester, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in *KDD* (1996), pp. 226–231
47. H. Wang Wei Fan, P.S. Yu, J. Han, Mining concept-drifting data streams using ensemble classifiers, in *KDD* (2003), pp. 226–235
48. B. Mozafari, H. Thakkar, C. Zaniolo, Verifying and mining frequent patterns from large windows over data streams, in *ICDE* (2008), pp. 179–188
49. A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, T. Seidl, Moa: massive online analysis, a framework for stream classification and clustering. *J. Mach. Learn. Res.* **11**, 44–50 (2010)
50. Y. Bai, C. Zaniolo, Minimizing latency and memory in DSMS: a unified approach to quasi-optimal scheduling, in *SSPS* (2008), pp. 58–67
51. Y. Bai, H. Thakkar, H. Wang, C. Zaniolo, Optimizing timestamp management in data stream management systems, in *ICDE* (2007), pp. 1334–1338
52. L. Golab, M. Tamer Özsu, Issues in data stream management. *ACM SIGMOD Rec.* **32**(2), 5–14 (2003)
53. D. Barbara, The characterization of continuous queries. *Int. J. Coop. Inf. Syst.* **8**(4), 295–323 (1999)
54. D.B. Terry, D. Goldberg, D.A. Nichols, B.M. Oki, Continuous queries over append-only databases, in *SIGMOD Conference* (1992), pp. 321–330
55. M. Sullivan, Tribeca: a stream database manager for network traffic analysis, in *VLDB* (1996), p. 594
56. S. Chandrasekaran et al., TelegraphCQ: continuous dataflow processing for an uncertain world, in *CIDR* (2003)
57. L. Liu, C. Pu, W. Tang, Continual queries for Internet scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng.* **11**(4), 583–590 (1999)
58. J. Chen, D.J. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for Internet databases, in *SIGMOD* (2000), pp. 379–390
59. H. Jagadish, I. Mumick, A. Silberschatz, View maintenance issues for the chronicle data model, in *PODS* (1995), pp. 113–124
60. A. Kumar Gupta, D. Suciu, Stream processing of xpath queries with predicates, in *SIGMOD Conference* (2003), pp. 419–430
61. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik, Monitoring streams—a new class of data management applications, in *VLDB*, Hong Kong, China (2002)
62. Y.-N. Law, H. Wang, C. Zaniolo, Relational languages and data models for continuous queries on sequences and data streams. *ACM Trans. Database Syst.* **36**, 8 (2011)
63. C. Luo, H. Thakkar, H. Wang, C. Zaniolo, A native extension of SQL for mining data streams, in *ACM SIGMOD Conference 2005* (2005), pp. 873–875

Hancock: A Language for Analyzing Transactional Data Streams

**Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers,
and Frederick Smith**

Massive transaction streams present a number of opportunities for data mining techniques. Transactions might represent calls on a telephone network, commercial credit card purchases, stock market trades, or HTTP requests to a web server. While historically such data have been collected for billing or security purposes, they are now being used to discover how the transactors, e.g. credit-card numbers or IP addresses, use the associated services.

For over six years, we have computed evolving profiles (called *signatures*) of the transactors in several large data streams. The signature for each transactor captures

C. Cortes, K. Fisher, D. Pregibon, A. Rogers and F. Smith, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 26 Issue 2, March 2004, Pages 301–338.

DOI: [10.1145/973097.973100](https://doi.org/10.1145/973097.973100), © 2004 ACM, Reprinted with permission.

C. Cortes · D. Pregibon
Google Research, 1440 Broadway, New York, NY 10018, USA

C. Cortes
e-mail: corinna@google.com

D. Pregibon
e-mail: daryl@google.com

K. Fisher (✉)
Computer Science Department, Tufts University, Medford, MA 02155, USA
e-mail: kfisher@eecs.tufts.edu

A. Rogers
University of Chicago, 1100 E 58th Street, Chicago, IL 60637, USA
e-mail: amr@cs.uchicago.edu

F. Smith
The Mathworks, 3 Apple Hill Drive, Natick, MA 01760, USA
e-mail: fsmith@mathworks.com

the salient features of his or her transactions through time. Programs for processing signatures must be highly optimized because of the size of the data stream (several gigabytes per day) and the number of signatures to maintain (hundreds of millions). Originally, we wrote such programs directly in C, but because signature programs often sacrificed readability for performance, they were difficult to verify and maintain.

Hancock is a domain-specific language created to express computationally efficient signature programs cleanly. In this chapter, we describe the obstacles to computing signatures from massive streams and explain how Hancock addresses these problems. For expository purposes, we present Hancock using a running example from the telecommunications industry; however, the language itself is general and applies equally well to other data sources.

1 Introduction

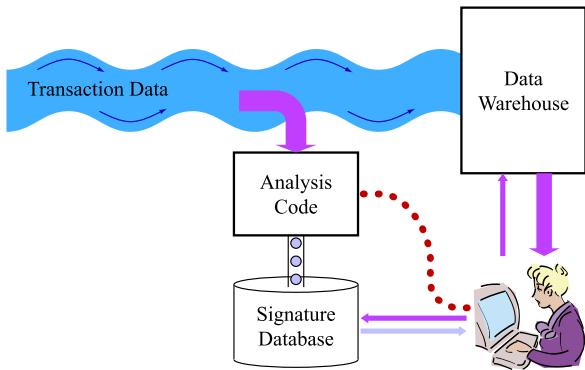
A transactional data stream is a sequence of records that log interactions between entities. For example, a stream of stock market transactions consists of buy/sell orders for particular companies from individual investors. Likewise, a stream of credit card transactions contains records of purchases by consumers from merchants. Data mining techniques are needed to exploit such transactional data streams because these streams contain a huge volume of simple records, any one of which is rather uninformative. When the records related to a single entity are aggregated over time, however, the aggregate can yield a detailed picture of evolving behavior, in effect, capturing the “signature” of that entity.

We have analyzed data streams from the telecommunications domain for more than six years. In our initial work, we processed roughly five million (M) international call-detail records per day, generated by approximately 12M accounts. In subsequent work, we have tackled larger and larger data streams, including the complete AT&T long distance data stream, which consists of approximately 300M records from roughly 100M accounts per day.

For each data stream, we compute or update signatures based on selected fields in each record in the data stream. A signature for a phone number might contain directly measurable features such as when most telephone calls are placed from that number, to what regions those calls are placed, and when the last call was placed. It might also contain derived information such as the degree to which the calling pattern from the number is “business-like” [12].

Figure 1 shows a fraud-detection application: a program taps a call-detail stream as it goes into a data warehouse and uses the tapped data to update the signatures of the telephone numbers with calls in the stream. If the new calls are inconsistent with the signature of a given telephone number, the system sends an alert to a network security representative. The representative then retrieves the relevant call-detail records from the warehouse to determine if the alert warrants further action. If the new calls are consistent with the existing signature, they are folded into the

Fig. 1 Typical use of a fraud signature



signature to allow a smooth evolution of calling behavior. Since most numbers exhibit consistent behavior through time, an overwhelming majority of calls are incorporated into the appropriate signatures and stored in the warehouse without being examined manually.

Programs to compute signatures must be highly optimized because of the size of the data stream and the number of signatures tracked. The size of the signature collection prevents us from keeping it entirely in memory. Consequently, signature programs are very I/O intensive: they must read from and write to the signature data on disk as they process transactions.

Our initial C programs for computing telecommunication signatures were efficient, but they often sacrificed readability to obtain this efficiency. Regulatory changes force frequent modifications to these programs. Consequently, program maintenance and verification, both of which require program readability, are important issues. When we started working with the complete AT&T long distance data stream, we realized that we needed software that could function at scale and yet be maintained as changes were required.

Hancock is a C-based domain-specific programming language that we designed and implemented in response to this need. By design, the language makes time and space efficient signature programs easy to read and write, independent of the quantity of data involved. Because Hancock manages scaling issues, it allows data analysts to experiment with new signatures quickly.

The goals of this chapter are to discuss the computational difficulties in writing efficient signature code for massive data streams, to show how Hancock alleviates these difficulties, and to discuss the motivations behind the Hancock design. Previous papers presented preliminary designs [5, 10], described the current design [11], and discussed implementation issues in depth [16].

2 Running Example

In this section, we describe the Cell Tower signature, which we will use as a running example to describe Hancock. This signature mines information from a wireless call

stream containing records used to bill accounts for calls sent and received from mobile telephones. Although these records contain many fields, only a few are relevant for computing the Cell Tower signature:

- Mobile Phone Number (MPN)
- Dialed telephone number
- First and last cell tower used

Our illustrative application characterizes the “diameter” of a mobile phone, i.e., is the phone used exclusively in one or a few neighboring cells, or is it used in a much larger region? Such information is useful for target marketing and for developing new service offerings.

To compute this information, we designed the Cell Tower signature. For each MPN, we track the five most frequently (and most recently) used cell towers and another value that captures the frequency with which calls placed from the MPN do not involve the top five cell towers. As one might expect, the top five list is dynamic, so the signature computation includes a probabilistic bumping algorithm that allows a new cell tower to enter the top five list as its frequency of use increases. Earlier work describes how to design signatures [6, 13–15].

Given the Cell Tower signature, any number of measures of “diameter” can be computed, e.g., the area of the convex hull defined by the geographical coordinates of the top five cell towers. Maintaining the list allows us to experiment with alternative measures before committing to any specific measure that might be computed directly from the call-detail stream.

As a prelude to subsequent sections in which we intersperse Hancock code with text, the following Hancock/C code describes the `profile` struct that we associate with each MPN in our Cell Tower signature:

```
#define NDIV 5
#define MAX_TOWER_NAME 12
typedef struct {
    char celltower[NDIV][MAX_TOWER_NAME];
    float freq[NDIV];
    float other;
} profile;
```

The `celltower` array stores the names of the five most commonly used towers as fixed-size C strings. The parallel array `freq` measures the frequency with which the corresponding tower is used. Field `other` measures how many calls are not reflected in the list of the top five towers.

3 The Hancock Language

Hancock makes it easier to read, write, and maintain signature programs by factoring into the language the issues that relate to scale. In this section we discuss some of these issues and describe how Hancock addresses them.

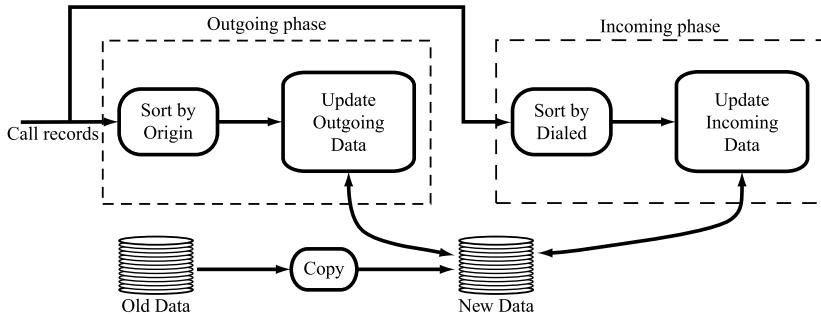


Fig. 2 High-level architecture of signature computations. The processing typically consists of several phases, each of which sorts the data in a different order and updates a different part of the signature

Prior to designing Hancock, we studied existing signature programs to understand their structure and the techniques they employ to achieve good performance. Figure 2 illustrates the process flow for a typical signature program from the telecommunications industry. Transaction records are collected for some time period, the length of which depends on the application (e.g., a day for marketing but just a few minutes for fraud detection). At the end of the time period, the records are processed to update the signatures. Before processing, the old signature data is copied to preserve a back-up for error-recovery purposes. During processing, the records are typically sorted in several ways, e.g., according to the originating and then the dialed phone numbers. After each sort, a pass is made over the data stream. During a pass, the portion of each signature relevant to the given sort is retrieved from disk, updated, and then written back to disk. For example, after sorting by the originating telephone number, only the portion of a signature that tracks out-bound calling would be updated typically; after sorting by the dialed number, the portion that tracks in-bound calling would be changed. Sorting the stream ensures good locality for accesses to the signatures on disk and groups the information relevant to each transactor into a contiguous segment of the stream.

3.1 Logical and Physical Streams

The fields in a transaction record are often encoded and packed to save space. In studying the original signature programs, we noticed that code to decipher the representation of records was interleaved with signature processing code, which made it difficult to respond to changes in the physical representation of records.

In Hancock, we separate the *physical* representation of the records in a data stream from the *logical* (expanded) representation on which we perform computations. This separation allows one person to understand the physical representation (the expert on that data source) but many people to use the logical representation (the consumers of that data source). This division facilitates maintenance: if the

physical representation changes, only the translation from the physical to the logical representation must be modified, presumably by the expert on that data source. The consumers need not modify their programs.

To declare a new stream type, programmers use the `stream` type operator. Generally, Hancock requires one stream type per data source. As there are many fewer data sources than there are signature programs, declaring new streams is rare. There are two forms of stream declarations: a specialized form for streams whose records are stored on disk in a fixed-width binary format, and a general form for records stored in other formats. The binary form is more convenient for the programmer, while the general form is more broadly applicable. In the general case, a stream declaration specifies a function that reads data from a file and returns a logical record. In the binary case, a stream declaration specifies both the physical and the logical representations for the records in the stream. It also specifies a function to convert from the encoded physical representation to the expanded logical representation.

The following binary stream declaration introduces the stream `wireless_s`:

```
stream wireless_s {
    getvalidWCR : wcrPhy_t => wcrLog_t;
};
```

For this stream of wireless call records, the C type `wcrPhy_t` serves as the physical representation and `wcrLog_t` serves as the logical. The identifier `getvalidWCR` names the function that specifies how to convert from the physical to the logical representation. This function, whose prototype is:

```
char getvalidWCR(wcrPhy_t *pc, wcrLog_t *c);
```

checks that the record `*pc` is valid and if so, unpacks `*pc` into `*c` and returns a constant to indicate a successful conversion. Otherwise, `getvalidWCR` simply returns a different constant to tell the system to skip to the next record.

Hancock provides *initializing declarations* to connect the on-disk representation of persistent data with Hancock variables. For example, the following code

```
wireless_s s = ``02-Dec-2004.calls'';
```

declares that the variable `s` has type `wireless_s` and can be found on disk at the location named by the path expression `“02-Dec-2004.calls”`.

In the remainder of the chapter, we use the term “record” to mean the logical representation of the elements in a stream, since stream definitions are the only place where the physical representation is needed. Figure 3 shows a sample stream of wireless call records.

C1	C2	C3	C4	C5	C6
973 5551212	973 5559898	973 5551313	973 5551212	973 5552222	973 5551212
201 5552323	773 5553344	201 5557777	773 5554545	201 5552323	773 5554545

Fig. 3 Portion of a wireless call stream running from left to right. Each box represents a call from an originating phone number to a dialed number. The first call, for example, was placed from (973) 555-1212 to (201) 555-2323. Each box is labeled with a call number (C1, for example). *White boxes* denote calls that originated from mobile phone numbers, while *dark gray/blue boxes* represent calls that originated from land lines

3.2 Logical, Approximate, and Physical Signatures

When the number of accounts is in the hundreds of millions, one often prefers to maintain only very small signatures for each account to reduce the I/O cost necessary to update the signatures. To save space, the values of a signature are often quantized or otherwise approximated before they are stored. For example, a floating point number representing the probability that a phone number is behaving like a business might be quantized into sixteen levels; similarly, the number of daily outbound minutes may be categorized according to one of eight logarithmically spaced usage bins. These approximate signatures can be compressed conveniently into a few bytes before writing them to disk. Thus, each signature program conceptually uses three different representations of each signature: the *logical* representation used for computation, the *approximate* representation that specifies what information to preserve, and the compressed *physical* form that is written to disk.

The original C signature programs contained routines to approximate and compress each signature before writing it to disk and routines to uncompress and expand it before computing with it. However, the original C code occasionally performed computations not only on the logical representations, but also on the approximate and on the compressed representations. While the code was very efficient, it was highly unreadable, making it difficult to verify and maintain.

Hancock's *view* construct provides a mechanism to specify two views of a single piece of data and the conversion between them. Signature programs use one view to describe the logical representation of each signature and another to describe the approximate representation. Hancock's *map* abstraction provides a mechanism to specify application-specific compression functions (see Sect. 4).

As an example of views, consider the following declaration that specifies approximate (*bin*) and logical (*minute*) representations of a unit of time:

```
view time(bin, minute) {
    char <=> int;
    bin(m) { return min_to_bin(m); }
    minute(b) { return bin_to_min(b); }
}
```

The line `char <=> int` declares that the *bin* view is represented as a `char` and the *minute* view is represented as an `int`. The *bin* function specifies how

to convert from the logical to the approximate representation by computing the bin associated with m minutes. Similarly, the `minute` function converts from the approximate to the logical representation by assigning a default number of minutes to the bin b . To translate between these two views, the programmer uses the Hancock view operator (\$):

```
bin b = 3;
minute m;
m = b$minute; // Convert bin b to minutes m
...
b = m$bin;     // Convert minutes m to
               // corresponding bin number
```

Views allow Hancock programmers to document the representation they are using in a given context. Views also ensure that the definition of how to convert between their representations appears only once in the program. Both of these aspects of views make Hancock programs easier to read and maintain than the corresponding C programs.

3.3 Signature Collections

The original signature programs used a data structure called a *map* to associate values with keys. Because performance requirements were tight, maps supported only two operations: associating a value with a key and retrieving the value associated with a key. To save disk space, maps were stored in a compressed format, customized for each application.

Hancock retained the notion of a map with a design that balances the need for performance on the one hand with the desire to separate the abstraction from the implementation on the other. We struck this balance by allowing programmers to tune map representations by specifying implementation parameters in map definitions. All subsequent uses of the map definition use only the abstract interface for maps: they do not depend upon the implementation parameters.

To make this discussion more concrete, consider the map declaration for the Cell Tower application, which associates a `profile` with each mobile phone number.

```
map cellTower_m {
    key 0..9999999999LL;
    split (10000, 100);
    value profile;
    default {{'\0', '\0', '\0', '\0', '\0'},
              {0.0, 0.0, 0.0, 0.0, 0.0},
              0.0};
};
```

The `key` clause indicates that the `cellTower_m` map will be indexed by values (of type `long long`) that range from 0 to 9999999999.

The `split` clause allows programmers to tune the implementation of maps by specifying `block` (10000) and `stripe` sizes (100). These parameters, which determine the decomposition of the key space in the implementation, are described in more detail in Sect. 4.

The `value` clause of a map declaration specifies the type of data to be associated with each key. The value type may be any valid Hancock type. In the Cell Tower application, this type is a standard C struct `profile`, but in many other applications the value type is the approximate representation from a Hancock view type. (In this signature application, the logical and approximate representations are the same.)

Finally, the `default` clause specifies a value to be returned if the programmer requests data for a key that does not have a value stored in the map. Such requests are relatively frequent because large transaction streams often contain data for *fresh* keys, for example, whenever a new telephone number, credit card number, or IP address is issued. Isolating default construction in map declarations allows code that queries a map for the value associated with a key to assume that it always receives a meaningful value, simplifying transaction processing code. For maps with type `cellTower_m`, the default profile contains empty strings for the cell towers and zeros for the frequencies.

As with streams, Hancock map variables can be connected to the associated on-disk representations using initializing declarations. In addition, such declarations can be annotated with qualifiers `const`, `exists`, and `new`, which we designed to protect valuable data. The `const` qualifier guards against writing to structures that should not change, `exists` ensures that the indicated location already has appropriate persistent data, while `new` guarantees that the indicated location is fresh.

Hancock supports a limited set of map operations that includes retrieving or updating the value associated with a key, iterating over a range of keys with stored values, and copying maps. Hancock maps do not support transactions, locking, secondary indices, or declarative querying to avoid the associated overhead.

Hancock programs access values in maps using an indexing operator `<: ... :>`. The code:

```
cellTower_m ct = ``02-Dec-2004.ct'';
pn_t mpn = 9735551212LL;
c = ct<:mpn:>;
...
ct<:mpn:> = c;
```

uses mobile phone number `mpn` to first read from and then write to map `ct`. A common idiom in Hancock

```
m<:key:>$logview
```

uses the indexing operator to get an approximate value out of a map (`m`) and the `view` operator to convert that value into the logical representation (`logview`).

In addition to the operations that manipulate individual items, Hancock also provides an operation that converts a map into a stream with one entry for each active key¹ from within a given range. The expression:

```
ct[startKeyExpr..stopKeyExpr]
```

generates a stream of active keys from map `ct` that fall between the values of the expressions `startKeyExpr` and `stopKeyExpr` inclusive.

Such map-to-stream expressions make it easy to write programs that update every value in a map or that evaluate queries that characterize the data. For example, the Cell Tower application uses map iteration to age each cell tower frequency each day, a process that allows new information to replace old information gradually. An auxiliary program for the Cell Tower application might generate a histogram that plots the number of MPNs with each possible number of recently-used cell towers by iterating over the map and assigning each active MPN to a histogram bucket based on its profile.

Finally, Hancock provides a lazy map copy operator, written using the infix notation `:=:`, to support coarse-grain roll-back for error-recovery. For example, the statement `new_ct :=: ct` initializes the map `new_ct` with the data from map `ct`.

3.4 Other Persistent Data

While building the first version of Hancock and porting the original production signatures, we observed the need for additional persistence mechanisms. In response, we added `directories` to group related information persistently and `pickles` to support custom persistent structures that could be split across memory and disk. We briefly describe the rationale behind the design of these mechanisms; see Cortes et al. [11] for more details.

We designed Hancock’s `directory` mechanism based on two related observations. First, map file names were often used to encode auxiliary information, such as the date of processing and the source of the data. Second, many applications used a collection of maps and other structures that together constituted the persistent data for the application. Grouping related information into a single unit allows programmers to dispense with arcane name conventions and be more confident that they will not improperly mix data from different time windows or applications.

The need for custom, partially memory-resident persistent data structures became apparent in studying the auxiliary data used in some signature applications. Some of this data did not fit comfortably into memory, and yet it was not a good candidate for the map abstraction. In response to this problem, we designed Hancock’s

¹An active key is one with an associated value stored in the map.

pickle mechanism, whichs allow programmers to design persistent structures with on-disk and in-memory representations tailored to the performance of individual signature programs. This mechanism is designed to integrate user-defined persistent data structures seamlessly with the other elements of Hancock’s persistent data system, in particular, with directories and initializing declarations.

3.5 Events

Much of the work in computing a signature is done in response to “events” in the input stream. For example, when a program sees a new mobile phone number in a `wireless_s` stream, it might re-initialize per-number counters. The original signature programs contained a hierarchy of events including seeing a new area code (`npa_begin`), seeing a new exchange (`nxx_begin`),² seeing a new phone number (`line_begin`), seeing an individual call record (`call`), seeing the last of a phone number (`line_end`), etc. Similar event hierarchies may be defined for streams of credit card charges, IP packets, etc. When a call-detail signature program detects an `npa_begin` event in a stream, it may retrieve the time zone for the triggering area code. In response to an `nxx_begin` event, it may retrieve all the old signatures for the newly seen exchange. For a `line_begin` event, it may initialize counters that it later increments in response to `call` events. The program may store the final values for these counters when a `line_end` event occurs. Analogous actions may be taken when processing other kinds of event hierarchies.

In Hancock, we divide this processing into two pieces: event detection and event response. Event detection includes defining the events of interest in a given stream and specifying how to identify them. Event response indicates what to do when an event is detected. The Hancock compiler generates the control flow that sequences the detection and response code. This control-flow code involves deeply nested loops that have an *inverted* control structure, which means that the code to respond to an ending event precedes the code for the corresponding beginning event. Hancock programs, which hide this structure, are much easier to read and maintain than the corresponding C programs, which mix the inverted control flow with event response code. In the remainder of this section, we discuss how to describe and detect events in a stream. In the next section, we discuss how to respond to detected events.

To define events in a general fashion, we introduced a new kind of type into Hancock: a *multi-union*. A multi-union names the set of labels it may contain and associates a type with each such label. Although we designed multi-unions to describe events, they are in fact a general construct, suitable for many purposes; hence we named their constituents “labels” instead of “events.” When we use multi-unions to describe events, however, we often refer to their labels as “events.” As an example, consider the declaration:

²An *exchange* is the first six digits of a ten digit telephone number.

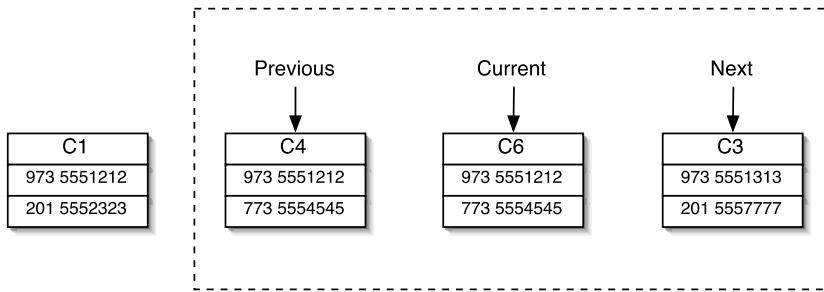


Fig. 4 Stream with window of type `*wcrLog_t[3:1]`

```
union line_e {:
    areacode_t npa_begin,
    exchange_t nxx_begin,
    pn_t      line_begin,
    wcrLog_t   call,
    pn_t      line_end,
    exchange_t nxx_end,
    areacode_t npa_end :};
```

This code creates a multi-union type `line_e` to describe typical events for call-detail streams. A value with this type contains any subset of the declared labels, including the empty set, which we write `{ : : }`. Each label in the set carries a value of the indicated type. If `l` is the current phone number and `c` the current call record in a stream, then the expression

```
{: line_begin = l, call = c :};
```

creates a value with type `line_e`. This value would describe the events that occur when the first (but not the last) call record for telephone number `l` appears in the stream. If `e1` and `e2` are multi-union values with the same type, then expression `e1:+:e2` produces a new value that contains the union of the labels of `e1` and `e2`.

After describing the events of interest using a multi-union declaration, the programmer must specify how to detect such events by writing an *event-detection* function. Such a function looks at a small portion of a stream and returns a multi-union to describe the events detected in that window.

To describe a small portion of a stream, Hancock provides a *window* type, illustrated in Fig. 4. The size of the window determines how many records in the stream can be viewed at once. A window is like an array, but has the added notion of a “current” record. In specifying a window, the programmer indicates the placement of the current record in the window. For example, the declaration:

```
wcrLog_t *w[3:1]
```

specifies that `w` is a window of size three onto a stream with records of type `wcrLog_t`. A pointer to the current record appears in the middle slot of the window, i.e., in `w[1]`. Slots with lower indices (`w[0]`) store pointers to records earlier in the stream; slots with higher indices (`w[2]`) look ahead to records appearing later in the stream. If the window overlaps either the beginning or the end of the stream (or both), the slots with no corresponding stream record are set to `NULL`.

An event detection function takes a window onto a stream and returns a multi-union describing the events detected for the current record in that window. The Cell Tower signature uses the `originDetect` function to process outgoing calls:

```
line_e originDetect(wcrLog_t *w[3:1])
{ line_e b,e;
  b = beginDetect(w[0], w[1]);
  e = endDetect(w[1], w[2]);
  return b :+: { : call = *w[1] :} :+: e;
}
```

This function calls the auxiliary functions `beginDetect` and `endDetect`. The first determines whether the current record represents a new MPN by comparing the `origin` from the previous record (`w[0]`) to the `origin` of the current record (`w[1]`). The second determines whether the current record represents the last call for a MPN by comparing the `origin` for the current record to the `origin` for the next record (`w[2]`).

3.6 Consuming a Stream

As in the original signature programs, Hancock's computation model is built around the notion of iterating over a sorted stream of transaction records. Sorting the records groups all the data relevant to one key into a contiguous segment of the stream and ensures good locality for map references that follow the sorting order. Consequently, each signature program typically makes multiple passes over its data stream. During each such pass, the signature program sorts the stream in a different order and updates a different portion of the signature associated with each key. We call each pass a *phase*.

We implement phases using Hancock's `iterate` statement, which has the following form:

C1	C4	C6	C3
973 5551212	973 5551212	973 5551212	973 5551313
201 5552323	773 5554545	773 5554545	201 5557777

begin_line(973 5551212) call(C1)

call(C4)

call(C6)

end_line(973 5551212)

begin_line(973 5551313)

call(C3)

end_line(973 5551313)

Fig. 5 Filtered, sorted stream labelled with events

```

iterate
  (over      stream variable
   filteredby filter predicate
   sortedby   sorting order
   withevents event detection function)
{
  event clauses
} ;

```

The header specifies an initial stream, a set of transformations to produce a new stream, and a function to detect events in the transformed stream. The body contains a set of event clauses that specify how to respond to the detected events.

We explain each of these pieces in turn. The `over` clause names the input stream. The `filteredby` clause specifies a predicate to drop unneeded records. For example, a `wireless_s` stream may include land-to-cell calls, which are not used by the Cell Tower signature. Immediately removing such records improves the efficiency of sorting and simplifies event response code. The `sortedby` clause describes a sorting order for the stream by listing the record fields that constitute the desired sorting key. For example, the clause

```
sortedby origin, connecttime
```

produces a stream sorted primarily by the originating telephone number and secondarily by the time at which the call was made. The `withevents` clause specifies an event detection function that computes the events triggered by the “current” record. Figure 5 shows a portion of a filtered, sorted, wireless call stream labeled with events.

The `event clauses` specify code to execute when an event detection function triggers an event. Events that occur simultaneously (i.e., in the same multi-union value) are processed in the order they appear in the event clauses. Given this ordering information, Hancock generates the control-flow to sequence the response code. The name of each event clause corresponds to a label in the multi-union returned by the event detection function. Each event clause takes as a parameter the value carried by the corresponding label. For example, the mobile phone number that triggers an `line_begin` event is passed to the `line_begin` event clause. The body of each event clause is a block of Hancock/C code.

As an example, Fig. 6 shows the outgoing phase of the Cell Tower signature, which processes the calls *made* by wireless telephone numbers. The function `out`,

```

1 void out(wireless_s calls, cellTower_m ct)
2 {
3     profile p;
4
5     iterate
6         ( over calls
7             filteredby completeCellCall
8             sortedby origin
9             withevents originDetect ) {
10
11     event nxx_begin(exchange_t npanxx) {
12         age(ct, npanxx);
13     }
14
15     event line_begin(pn_t mpn) {
16         initProfile(&p);
17     }
18
19     event call(wcrLog_t c) {
20         aggregate(&p, c.cellid);
21     }
22
23     event line_end(pn_t mpn) {
24         profile old;
25         old = ct <:mpn:>;
26         ct <:mpn:> = update(&old,&p);
27     }
28 };
29 }
```

Fig. 6 Outgoing phase for the Cell Tower signature

which encapsulates this phase, contains a single iterate statement that processes a wireless call stream. It uses the predicate function `completeCellCall` to remove incomplete and non-cellular calls from the stream. It sorts the filtered stream by the originating phone number. It uses the function `originDetect` to find events in the sorted stream. The event clauses in lines 11 to 27 of Fig. 6 specify how to respond to the detected events, aging all the signatures in an exchange with the function `age`, initializing a temporary profile to track the calls for a given phone number with the function `initProfile`, integrating each call into the temporary profile with the function `aggregate`, and finally updating the map with the day's profile. Note that this phase does not use all the events defined in the `line_e` type.

3.7 Putting It Together

In the previous section, we explained that computing a signature may require multiple passes over the data. Hancock provides the `sig_main` construct to express the

data flow between such passes: the arcs between the phase boxes in Fig. 2 depict this construct. In addition, the `sig_main` function indicates the entry-point into Hancock programs, provides a simple way for programmers to specify command-line arguments, and augments initializing declarations as a way to connect persistent representations with program variables. The Hancock compiler generates code to parse the actual command-line parameters, relieving the programmer of this tedious task.

As an example, consider the `sig_main` function for the Cell Tower signature:

```
void sig_main(const wireless_s calls <c:>,
              exists const cellTower_m oldCT <m:>,
              new cellTower_m newCT <M:>) {
    newCT :=: oldCT;
    out(calls, newCT);
}
```

The `calls` parameter is a stream that contains the raw wireless call data. The syntax (`<c:>`) after the variable name specifies that this parameter will be supplied as a command-line option using the `-c` flag. The colon indicates that the flag takes an argument, in this case the path to the on-disk representation of the wireless stream. The absence of a colon indicates that the parameter is a boolean flag. The `oldCT` parameter is an existing Cell Tower map, the location of which is specified using the `-m` flag. The `newCT` parameter names the Cell Tower map to hold the result of this program. The `-M` flag specifies the location for this map. The qualifiers on `sig_main` parameters have the same meaning as qualifiers in initializing declarations.

In general, the body of `sig_main` is a sequence of Hancock and C statements. In the Cell Tower application, `sig_main` copies the data from `oldCT` into `newCT` and then invokes the outgoing phase with the raw call stream and the new Cell Tower map as arguments. If the Cell Tower application required a second phase, e.g., an incoming phase, we would call it after the call to `out`.

4 Implementation

In this section, we give a brief overview of our implementation of Hancock. The Hancock compiler translates Hancock code into C and then invokes a platform-dependent compiler to convert the resulting code into an object file. That object file is then linked to the Hancock runtime system to produce an executable.

To implement the translation, we modified CKIT [8], a C-to-C translator written in ML, to parse Hancock and translate the resulting extended parse tree into abstract syntax for C. During the translation, we typecheck the various Hancock forms, which allows us to report errors in terms of the Hancock source code, rather than in terms of the resulting C code. The translator generates code to implement command-line processing, directories, and the `iterate` statement, which significantly reduces the amount of code the programmer needs to write. After translation, we use the CKIT pretty-printer to produce C code.

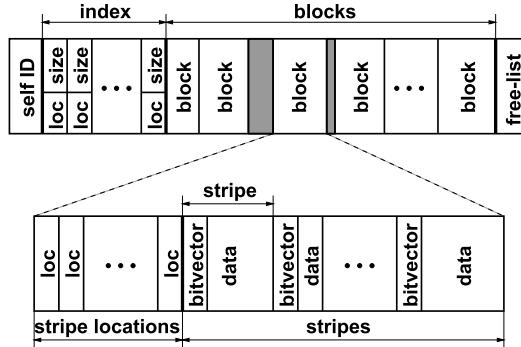


Fig. 7 The on-disk data representation of a Hancock map includes a self-identification field, a block index, an unordered set of blocks, and a free-list for unused space. The self-identification information is used to match the file against the declared type of a map to help prevent users from inadvertently using a map with the wrong type for an application. Each block contains a stripe index followed by the stripes, in order

The runtime system, written in C, mediates access to persistent data. It converts between the representation of data on-disk and in memory as necessary. Managing stream, directory, and pickle data is straightforward: the runtime system provides the necessary I/O and calls the appropriate user or compiler-generated functions.

Managing map data is more complex because it requires a data structure that satisfies the temporal requirements of our applications without introducing significant space overhead. Hancock's map implementation uses the multi-level table³ shown in Fig. 7 augmented with compression to reduce disk space requirements. To find the location of a key in the table, the runtime system splits keys into three pieces: a block number, a stripe number, and an entry number. It uses these pieces to index into different levels of the table. The top level table maps a block number to its location in memory or on disk. Each block contains a stripe index that maps a stripe number to its starting location within the block. Once the location of the relevant stripe is identified, the runtime system reads in the stripe, which contains a bitvector followed by compressed data,⁴ and decompresses it. Finally, the entry number determines the location of the key within the decompressed stripe.

Recall that programmers specify the block size and the stripe size for a map using the `split` clause in type declaration. The main tradeoff to consider when deciding the block size is the size of the index (12 bytes per entry on disk, 20 in memory). Choosing a stripe size, on the other hand, requires weighing various considerations, including the cost of decompressing values, the size of a decompressed stripe in relation to the processor's primary and secondary cache sizes, and the expected mix

³Other applications, such as paging or IP address lookup for routing [17, 19, 21] use variants of a multi-level table to support large key spaces.

⁴Hancock supplies default compression functions, but programmers can also specify application-specific compression routines as part of a map type declaration.

of access patterns for the data. See Fisher et al. [16] for a more detailed discussion of these tradeoffs.

5 Experiences

In this section, we describe our experiences with Hancock programs in practice. We rewrote the original domestic long-distance signature programs in Hancock. These programs, which have been running in production every day for more than six years, are as time efficient as the original C programs, while at the same time being more space efficient, shorter, and better organized. The readability of the Hancock programs makes them easy to check for compliance with (frequently changing) federal legislation. The Hancock programs are also easy to update in response to changes in the transaction data. For example, updating the programs to make them aware that area code 866 and 877 had become toll-free required changing only two lines of one header file and recompiling the programs.

In addition to supporting the original applications better, Hancock’s domain-specific abstractions and improved performance enabled data analysts to craft new kinds of signatures. When analysts used C directly, they were able to store only two-to four-byte signatures. Because of this limited space, they had to use very rough approximations. Although this rough data was very useful for certain kinds of marketing applications, it was not suitable for many other kinds of applications, notably fraud detection. Hancock’s abstractions and their efficient performance enabled analysts to build signatures containing over 100 bytes. With that level of detail, analysts were able to store sufficiently precise information to enable applications previously thought to be infeasible.

Most existing Hancock programs manipulate long-distance call-detail data because the data analysts we work with focus on that domain. However, nothing in Hancock is specific to this domain; Hancock gives programmers full control over the description of their data sources. Programmers have used Hancock to analyze data from various sources: wireless call records, calling-card call records, telephone numbers, TCPDUMP data, IP addresses, and even referee reports.

6 Related Work

Broadly speaking, there are two different classes of work related to Hancock: work on persistent data structures and work on stream processing.

6.1 Persistent Data

Many languages provide support for persistent data. This support typically falls into one of three categories: pickle-based approaches [1, 23, 24, 27, 29], interfaces to

databases, and orthogonal persistence systems [2, 20, 22]. Pickles provide a way to convert a value into a stream of bytes and vice versa, supporting persistence, but not in a way that allows data structures to reside partially in-memory and partially on-disk, a requirement for very large data. A second common approach to supporting persistence is to provide an interface to a standard relational or object-oriented database. We rejected this approach for Hancock because we did not believe that a traditional database could handle the high-percentage of updates generated during daily stream processing [3, 4, 7, 16, 18, 26]. Orthogonal persistence systems automatically determine if a given piece of data must be persistent by starting from a collection of persistent roots and making all reachable data persistent. This approach is akin to automatic memory management, which can simplify programming, but at some performance cost. Because of the tight space and time requirements of our domain, we adopted a more explicit technique for persistence in Hancock.

6.2 Stream Processing

Many systems support stream operations; here we describe systems designed to handle high-volume stream data. The oldest of these systems, Tribeca [26], pre-dated Hancock. More recently, high-volume stream processing has become an area of active interest in the database community [25, 28]. Aurora [7], Telegraph [18], and STREAMS [3] are all examples of systems under development for computing with high-volume streams. In contrast to Hancock, which has been deployed in production for several years, these newer systems are in various stages of prototyping. Further experience is necessary to determine how well these systems will scale. We briefly describe the focus of each of these projects.

Tribeca [26] is a system for monitoring network traffic. It provides a query language with operations for separating and recombining streams, operations for computing moving-aggregates over windows, and a restricted form of join. The separation and recombination operators might be used, for example, to convert a packet stream into a session stream. Tribeca provides more support for describing and manipulating streams than Hancock does, but it provides less support for computing with the individual elements in a stream or for integrating with persistent data.

Aurora [7] and Hancock are complementary. Aurora is a system designed to monitor stream data. It supports queries over multiple streams of data and allows queries to join and leave the system over time. One can view the queries combined with the streams as a graph. At the end of any path in the graph is an application that consumes the resulting data; that application could be a Hancock program.

Telegraph [18] is an adaptive dataflow system designed to compute continuous queries over streams of data. PSoup [9], a system built on top of Telegraph, expands upon this model to allow the query mix to change over time. This system can be used to compute aggregates from stream data, such as how many music downloads occurred in a given subnet within the last hour. Like Aurora, Telegraph is not designed to provide direct support for integrating stream data into persistent structures, the essential operation in computing signatures.

The members of the STREAMS project [3] are developing a system for executing continuous queries over multiple streams. The focus of this project is to develop fundamental models of stream data systems and efficient methods for managing resources in such systems. At present, their model explicitly excludes queries that can modify persistent data during computation. This restriction, which may be removed over time, eliminates signatures as a possible application for STREAMS.

7 Language Versus Library

One question we are asked often is why we chose to design a language rather than a library. There are two technical reasons for choosing the language option. First, expressing Hancock's event model and the information sharing it provides proved awkward in a call-back⁵ framework, the usual technique for implementing such abstractions. Second, by designing a language we could use the language's type system to provide more precise typechecking than is provided by C. For example, the natural way to implement maps using a library interface would require the programmer to cast between the actual type of a value and `void *`, thereby losing the benefits of typechecking. The scale of the data makes the complexity of finding and fixing bugs in signature programs substantial. Therefore, static error detection is essential.

The more compelling reason to choose a language over a library for us is sociological. The experience of writing a Hancock program is fundamentally different than writing the equivalent program in C. This difference arises in part because Hancock removes issues of scale, leaving programmers free to concentrate on the design of the individual profiles, and in part because Hancock provides a vocabulary tailored to the domain of signature design.

8 Conclusions

Working with transactional data streams is like drinking from the proverbial fire hose: the volume is simply overwhelming. But this challenge provides an opportunity for data mining research to enter a new area. We believe that Hancock is a valuable tool for exploiting this opportunity.

Hancock has allowed us to improve our application base by replacing hard-to-maintain, hand-written C code with disciplined Hancock code. Because Hancock provides high-level, domain-specific abstractions, Hancock programs are easier to read and maintain than the earlier C programs. By careful design, these abstractions have efficient implementations, which allow Hancock programs to preserve the execution speed and data efficiency of the earlier C programs. Hancock gave domain

⁵A call-back is a call from a function in a library "back" to a function in user code.

experts the confidence to attack more challenging problems because it allowed them to concentrate on *what* to compute without worrying about *how* to manage the volume of data.

Hancock is publicly available for non-commercial use from:

www.research.att.com/projects/hancock.

References

1. A.W. Appel, A runtime system. *Lisp and Symbolic Computation* **4**(3), 343–380 (1990)
2. M. Atkinson, L. Daynes, M. Jordan, T. Printezis, S. Spence, An orthogonally persistent Java. *ACM SIGMOD Rec.* **25**(4) (1996)
3. B. Babcock, S. Babu, M. Data, R. Motwani, J. Widom, Models and issues in data stream systems, in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems (PODS 2002)* (2002). See the Stream Project homepage, www-db.stanford.edu/stream for a complete list of papers
4. D. Belanger, K. Church, A. Hume, Virtual data warehousing, data publishing, and call detail, in *Proceedings of Databases in Telecommunications 1999, International Workshop*. Also Appears in Springer Verlag LNCS, vol. 1819 (1999), pp. 106–117
5. D. Bonachea, K. Fisher, A. Rogers, F.S. Hancock, A language for processing very large-scale data, in *USENIX 2nd Conference on Domain-Specific Languages*, USENIX Association (1999), pp. 163–176
6. P. Burge, J. Shawe-Taylor, Frameworks for fraud detection in mobile telecommunications networks, in *Proceedings of the Fourth Annual Mobile and Personal Communications Seminar*, University of Limerick (1996)
7. D. Carney, U. Cetinmehl, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik, Monitoring streams—a new class of data management applications, in *Proceedings of the 28th VLDB Conference* (2002). See the Aurora Project homepage, www.cs.brown.edu/research/aurora/main.html for a complete list of papers
8. S. Chandra, N. Heintze, D. MacQueen, D. Oliva, M. Siff, Pre-release of C-frontend library for SML/NJ (1999). See cm.bell-labs.com/cm/cs/what/smlnj
9. S. Chandrasekaran, M.J. Franklin, Streaming queries over streaming data, in *Proceedings of the 28th VLDB Conference* (2002)
10. C. Cortes, K. Fisher, D. Pregibon, A. Rogers, F.S. Hancock, A language for extracting signatures from data streams, in *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining* (2000), pp. 9–17
11. C. Cortes, K. Fisher, D. Pregibon, A. Rogers, F.S. Hancock, A language for analyzing transactional data streams. *ACM Transactions on Programming Languages and Systems* **26**(2), 301–338 (2004)
12. C. Cortes, D. Pregibon, Giga mining, in *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining* (1998)
13. C. Cortes, D. Pregibon, Information mining platform: an infrastructure for KDD rapid deployment, in *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining* (1999)
14. D.E. Denning, An intrusion-detection model, *IEEE Trans. Softw. Eng.* **13**(2) (1987)
15. T. Fawcett, F. Provost, Adaptive fraud detection. *Data Mining and Knowledge Discovery* **1**, 291–316 (1997)
16. K. Fisher, C. Goodall, K. Hogstedt, A. Rogers, An application-specific database, in *Proceedings of 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01)*. LNCS, vol. 2397 (Springer, Berlin, 2002), pp. 213–227

17. P. Gupta, S. Lin, M. McKeown, Routing lookups in hardware and memory access speeds, in *Proc. 17th Ann. Joint Conf. of the IEEE Computer and Communications Societies*, vol. 3 (1998), pp. 1240–1247
18. J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, M. Shah, Adaptive query processing: technology in evolution, in *IEEE Data Eng. Bulletin* (2000), pp. 7–18. See the Telegraph Project homepage telegraph.cs.berkeley.edu for a complete list papers
19. N.-F. Huang, S.-M. Zhao, J.-Y. Pan, C.-A. Su, A fast IP routing lookup scheme for gigabit switching routers, in *Proc. 18th Ann. Joint Conf. of the IEEE Computer and Communications Societies*, vol. 3 (1999), pp. 1429–1436
20. M. Knasmüller, Adding persistence to the Oberon system, in *Proceedings of the Joint Modular Languages Conference 97* (1997)
21. B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking* 7(3), 324–334 (1999)
22. B. Liskov, M. Castro, L. Shrira, A. Adya, Providing persistent objects in distributed systems, in *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)* (1999)
23. G. Nelson (ed.), *Systems Programming with Modula-3* (Prentice Hall, New York, 1991)
24. R. Riggs, J. Waldo, A. Wollrath, K. Bharat, Pickling state in the Java system, in *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)* (1996)
25. SIGMOD. Proceedings of SIGMOD (2002)
26. M. Sullivan, A. Heybey, Tribeca: a system for managing large databases of network traffic, in *Proceedings of the USENIX Annual Technical Conference (No. 98)* (1998)
27. G. van Rossum Python library reference (2001). python.sourceforge.net-devel-docs/lib/lib.html
28. VLDB. Proceedings of the 28th VLDB conference (2002)
29. D.C. Wang, The asdlGen reference manual. See www.cs.princeton.edu/zephyr/ASDL (1998)

Sensor Network Integration with Streaming Database Systems

Daniel Abadi, Samuel Madden, and Wolfgang Lindner

1 Introduction

Recent advances in computing technology have led to the production of a new class of computing device—the wireless, battery powered, smart sensor [30]. Traditional sensors deployed throughout buildings, labs, and equipment are passive devices that simply modulate a voltage based on some environmental parameter. In contrast, these new sensors are active, full-fledged computers, capable not only of sampling real world phenomena but also filtering, sharing, and combining those sensor readings with each other and nearby Internet-equipped endpoints.

Smart-sensor technology has enabled a broad range of ubiquitous computing applications [13]: the low cost, small size, and untethered nature of these devices makes it possible to sense information at previously unobtainable resolutions. Animal biologists can monitor the movements of hundreds of different animals simultaneously, receiving updates of location as well as ambient environmental conditions every few seconds [8, 25]. Vineyard owners can place sensors on every one of their plants, providing an exact picture of how various light and moisture levels vary in the *microclimates* around each vine [7]. Supervisors of manufacturing plants,

D. Abadi

Department of Computer Science, Yale University, 51 Prospect Street, New Haven, CT 06511,
USA

e-mail: dna@cs.yale.edu

S. Madden (✉) · W. Lindner

Computer Science and Artificial Intelligence Laboratory (CSAIL),
Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA 02139, USA
e-mail: madden@csail.mit.edu

W. Lindner

e-mail: wolfgang@csail.mit.edu

temperature controlled storage warehouses, and computer server rooms can monitor each piece of equipment, and automatically dispatch repair teams or shutdown problematic equipment in localized areas where temperature spikes or other faults occur.

Over the past several years, we have designed and implemented a query processor for such sensor networks called TinyDB (for more information on TinyDB, see the TinyDB Home Page [24]). TinyDB is a distributed query processor that runs on each of the nodes in a sensor network that is explicitly designed to simplify many of the data collection applications described above. TinyDB runs on the Berkeley *mote* platform, on top of the TinyOS [17] operating system. TinyDB has many of the features of a traditional query processor (e.g., the ability to select, join, project, and aggregate data), but also incorporates a number of other optimization features designed to minimize power consumption.

In this chapter, we review of many of the features of TinyDB and other sensor network query processing systems (such as Cougar [41]). We also discuss how TinyDB interfaces with a data stream management system (DSMS), examples of which are discussed elsewhere in this book. Such integration is important a number of reasons, including:

1. Integration provides the ability to combine stored or streaming data from the DSMS with data from the sensornet. For example, users may want to compute a join to decide if readings from a motion-detector are correlated with network activity, or if truck locations match the expected locations on the planned route.
2. Integration provides a single, integrated interface for interacting with both the streaming database and the sensor network. TinyDB currently allows users to log query results into a relational database table via JDBC, but queries must still be input to TinyDB independently of the relational database. By providing a single, seamless system, users are only require knowledge of configuration and interaction with one set of interfaces.
3. Integration offers the ability to optimize between the database system and the sensor network. For example, it may be desirable to push certain filters and aggregates into the sensor network if user queries are interested in only particular subsets or coarse summaries of readings.

To provide this integration, we interpose a *proxy* between the DSMS and TinyDB. This proxy selects fragments of a query input at the DSMS that can be safely and efficiently run on the sensor network, and facilitates optimization between the two query processing systems. It also hides from the DSMS the sensor-network specific communication and control protocols, and prevents the sensor network from needing to provide a complete set of query processing operators or sophisticated query parsing or optimization features.

Before covering the details of this proxy system, however, we begin with a brief discussion of the TinyOS operating system and the mote hardware upon which TinyDB is built.

2 Sensor Networks and TinyOS

A sensor network node is a battery-powered, wireless computer. Typically, these nodes are physically small (a few cubic centimeters) and extremely low power (a few tens of milliwatts versus tens of watts for a typical laptop computer). Power is of utmost importance. If used naively, individual sensor nodes will deplete their energy supplies in only a few days.¹ In contrast, if sensor nodes are very spartan about power consumption, months or years of lifetime are possible. Mica motes, for example, when operating at 2 % duty cycle (between active and sleep modes) can achieve lifetimes in the 6 month range on a pair of AA batteries. This duty cycle limits the active time to 1.2 seconds per minute.

There have been several generations of motes produced. Older, *Mica* motes have a 4 MHz, 8 bit Atmel microprocessor. Their RFM TR1000 [33] radios run at 40 kbytes/s over a single shared CSMA/CA (carrier-sense multiple-access, collision avoidance) channel. Newer *Mica2* nodes use a 7 MHz processor and a radio from ChipCon [10] corporation which runs at 38.4 kbytes/s. A third mote, called the *Mica2Dot* has similar hardware as the *Mica2* mote, but uses a slower, 4 MHz, processor. Pictures of a *Mica* and *Mica2Dot* mote are shown in Fig. 1. *Mica* motes are visually very similar to *Mica2* motes and are exactly the same form factor. Newer motes, from Crossbow, Telos, and other companies, still use low power, low-frequency microprocessors with very limited RAM (such as the TI MSP430 series, which has up to 10 KB of RAM and runs up to 16 MHz), but now typically use ZigBee 802.15.4 radios with bit rates up to 250 kbytes/s (such as the ChipCon 2420).

Radio messages are in TinyOS variable size. Typically about 20 50-byte messages (the default size in TinyDB) can be delivered per second. Like all wireless radios (but unlike a shared EtherNet [11], which uses the collision detection (CD) variant of CSMA), both the RFM and ChipCon radios are half-duplex, which means that they cannot detect collisions because they cannot listen to their own traffic. Instead, they try to avoid collisions by listening to the channel before transmitting and backing off for a random time period when it is in use.

2.1 TinyOS

Berkeley provides a primitive OS for the motes called TinyOS. TinyOS consists of a set of components for managing and accessing the mote hardware, and a “C-like” programming language called nesC. TinyOS has been ported to a variety of hardware platforms, including UC Berkeley’s Rene, Dot, Mica, Mica2, and Mica2Dot motes, the Blue Mote from Dust Inc. [18], and MIT’s Cricket [32] platform.

¹At full power, a Berkeley *Mica* mote (see Fig. 1) draws about 15 mA of current. A pair of AA batteries provides approximately 2200 mAh of energy. Thus, the lifetime of a *Mica2* mote will be approximately $2200/15 = 146$ hours, or 6 days. *Mica2Dot* motes, with smaller 800 mAh batteries and peak current consumption of 12 mA can run at full power for just $800/12 = 66$ hours, or about 3 days.

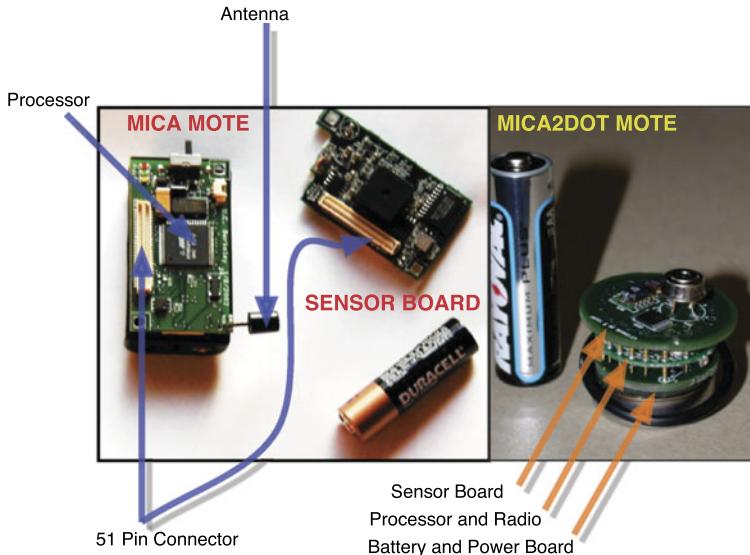


Fig. 1 Two Berkeley motes

The major features of TinyOS are:

1. A suite of software designed to simplify access to the lowest levels of hardware in an energy-efficient and contention-free way, and
2. A programming model and the nesC language designed to promote extensibility and composition of software while maintaining a high degree of concurrency and energy efficiency. Interested readers should refer to [15].

It is interesting to note that TinyOS does not provide the traditional operating system features of process isolation or scheduling (there is only one application running at time), and does not have a kernel, protection domains, memory manager, or multi-threading. Indeed, in many ways, TinyOS is simply a library that provides a number of convenient software abstractions, including:

- The radio stack, which sends and receives packets over the radio and manages the MAC layer [17, 39].
- Software to read sensor values—each sensor device (e.g., the light sensor) is managed by a software component that provides commands to fetch a new reading from the sensor, and possibly access calibration and configuration features for more sophisticated digital sensors.
- Components to synchronize the clocks between a group of motes and schedule timers to fire at specified times [12].
- Power management features that allow an application to put the device into a low power sleep mode without compromising the state of any other software components.
- Software to manage the off-chip Flash using a simple, file-system like interface.

- Components to localize a sensor relative to neighboring node by emitting sound or ultrasound pulses and measuring their time-of-flight to those neighbors [38].

Thus, TinyOS and nesC provide a useful set of abstractions on top of the bare hardware. Unfortunately, they do not make it particularly easy to author software for many of the data collection applications discussed in the introduction. Sensor networks will never be widely adopted if every application requires this level of engineering effort. The declarative model we advocate reduces these applications to a few short statements in a simple language.

3 TinyDB

In TinyDB, queries are input at the user’s PC in a simple SQL-like language which describes the data the user wishes to collect and ways in which he or she would like to combine, transform, and summarize it. The most significant way in which the variant of SQL we have developed differs from traditional SQL is that queries are *continuous* and *periodic*. That is, users register an interest in certain kinds of sensor readings (e.g., “temperatures from sensors on the 4th floor every 5 seconds”) and the system streams these results out to the user. We call each period in which a result is produced an *epoch*. The *epoch duration*, or *sample period*, of a query refers to the amount of time between successive samples; for this example, the sample period would be 5 seconds. As we discuss various aspects of our system, we will show some examples of our language syntax and discuss its other features (new and in common with traditional SQL) in more detail.

In TinyDB, query optimization is done as much as possible on the server-side PC, since it can be quite computationally intensive. However, because the server may not have perfect state about the status of the sensor network, and because costs used to optimize a query initially may change over its lifetime, it is sometimes necessary to *adapt* running query plans once they have been sent into the network.

3.1 Query Language

Queries in TinyDB, as in SQL, consist of SELECT–FROM–WHERE–GROUPBY–HAVING blocks supporting selection, join, projection, aggregation, and grouping. TinyDB also includes explicit support for windowing and subqueries via *materialization points*. In queries, we view sensor data as a single virtual table with one column per sensor type. Tuples are appended to this table periodically, at well-defined intervals that are a parameter of the query. This period of time between each sample interval is the *epoch*, as described above. Epochs provide a convenient mechanism for structuring computation to minimize power consumption. As an example, consider the query

```

SELECT nodeid, light, temp
  FROM sensors
SAMPLE PERIOD 1s FOR 10s

```

This query specifies that each sensor should report its own id, light, and temperature readings once per second for 10 seconds. The virtual table `sensors` contains one column for every attribute available in the catalog and one row for every possible instant in time. The term *virtual* means that these rows and columns are not actually materialized—only the attributes and rows referenced in active queries are actually generated.

Results of this query stream out of the network fashion where they may be logged, output to the user, or fed into another database system. The output consists of an ever-growing sequence of tuples, clustered into 1 s time intervals. Each tuple includes a time stamp corresponding to the time it was produced.

Note that the `sensors` table is (conceptually) an unbounded, continuous *data stream* of values; as is the case in other streaming and online systems, certain blocking operations (such as sort and symmetric join) are not allowed over such streams unless a bounded subset of the stream, or *window*, is specified. Windows in TinyDB are defined as fixed-size materialization points over the sensor streams. Such materialization points accumulate a small buffer of data that may be used in other queries. Consider, as an example, the following query:

```

CREATE
  STORAGE POINT recentlight SIZE 8 seconds
  AS (SELECT nodeid, light FROM sensors
      SAMPLE PERIOD 1s)

```

This statement provides a shared, local (i.e., single-node) location to store a streaming view of recent data similar to materialization points in other streaming systems like Aurora or STREAM [2, 26], or materialized views in conventional databases.

Joins are allowed between two storage points on the same node, or between a storage point and the `sensors` relation, in which case `sensors` is used as the outer relation in a nested-loops join. That is, when a `sensors` tuple arrives, it is joined with tuples in the storage point at its time of arrival. This is effectively a *landmark query* [16] common in streaming systems. Consider, as an example

```

SELECT COUNT(*)
  FROM sensors AS s, recentLight AS rl
 WHERE rl.nodeid = s.nodeid
   AND s.light < rl.light
SAMPLE PERIOD 10s

```

This query outputs a stream of counts indicating the number of recent light readings (from 0 to 8 samples in the past) that were brighter than the current reading.

TinyDB also includes support for grouped aggregation queries. Aggregation has the attractive property that it reduces the quantity of data that must be transmitted

through the network, and thus can reduce energy consumption and bandwidth usage by replacing more expensive communication operations with relatively cheaper computation operations, extending the lifetime of the sensor network significantly. TinyDB also includes a mechanism for user-defined aggregates and a metadata management system that supports their optimization.

In addition to aggregates over values produced during the same sample interval (for example, as in the COUNT query above), users want to be able to perform temporal operations. For example, in a building monitoring system for conference rooms, users may detect occupancy by measuring maximum sound volume over time and reporting that volume periodically:

```
SELECT WINAVG(volume, 30s, 5s)
  FROM sensors
 SAMPLE PERIOD 1s
```

This query will report the average volume over the last 30 seconds once every 5 seconds, acquiring a sample once per second. This is an example of a *sliding-window* query common in many streaming systems [16, 26].

When a query is issued in TinyDB, it is assigned an identifier (id) that is returned to the issuer. This identifier can be used to explicitly stop a query via a “STOP QUERY *id*” command. Alternatively, queries can be limited to run for a specific time period via a FOR clause, or can include a stopping condition as a *triggering condition* or *event*; see our recent work on *acquisitional query processing* [23] for more detail about these language constructs.

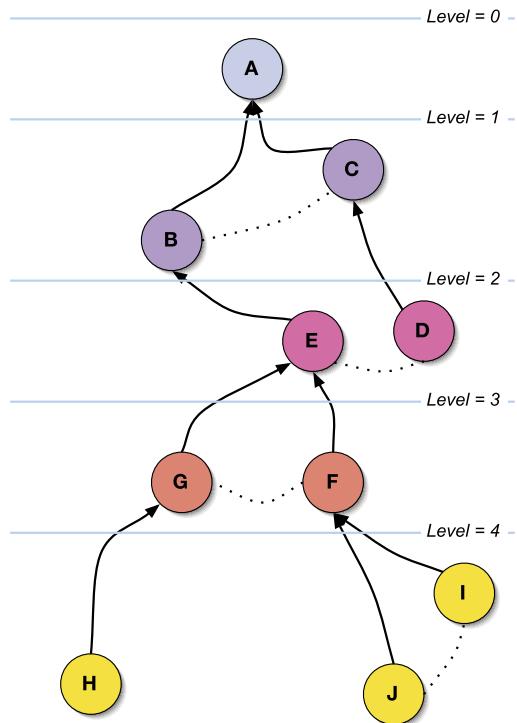
3.2 Query Dissemination and Result Collection

Once a query has been optimized, it is *disseminated* into the network. We discuss one basic communication primitive, a *routing tree*. A routing tree is rooted at either the base station or a storage point and it allows the root of the network to disseminate a query and to collect query results. This routing tree is formed by forwarding the query from every node in the network: the root initially transmits the query; all *child* nodes that hear it process it and forward it on to their children, and so on, until the entire network has heard about the query.

Each radio message contains a hop-count, or *level*, indicating the distance from the broadcaster to the root. To determine their own level, nodes pick a *parent* node that is (by definition) one level closer to the root than they are. This parent will be responsible for forwarding the node’s (and its children’s) query results to the base station. We note that it is possible to have several routing trees if nodes keep track of multiple parents. This can be used to support several simultaneous queries with different roots. This type of communication topology is common within the sensor network community and is known as *tree-based routing*.

Figure 2 shows an example sensor network topology and routing tree. Solid arrows indicate parent nodes, while dotted lines indicate nodes that can hear each

Fig. 2 A sensor network topology, with routing tree overlay



other but do no use each other for routing. In general, a node may have several possible choices of parent; a simple approach is to chose the parent to be the ancestor node with the lowest level. In practice, it turns out that making a proper choice of parent is quite important in terms of communication and data collection efficiency and that network topologies are much less regular and more complex than one might expect [14]. Unfortunately, the details of the best known techniques forming trees in real networks are quite complicated and outside the scope of our discussion in this paper. For a more complete discussion of these and other issues, see, for example, recent work from the TinyOS group at UC Berkeley [40]. We would like to note that there has been a plethora of work on routing in ad-hoc and sensor networks [19, 21, 27, 28], including energy-aware routing [9, 31] and special MAC protocols [42]. Our goal here is different: instead of a general-purpose routing layer, we need to disseminate information from sensors to the root, and we can leverage this knowledge about our communication patterns.

Once a routing tree has been constructed, each node has a connection to the root of the tree which is just a few radio hops long. We can then use this tree to collect data from sensors by having them forward query results up this path. In TinyDB, the routing tree evolves over time as new nodes come online, interference patterns change, or nodes run out of power. Tree maintenance is done locally, at every node, by keeping a set of candidate parents and an estimate of the quality of the communication link with each of them; when the quality of the link to the current

parent is sufficiently worse than the quality to another candidate parent, a switch is made.

A simple routing structure such as routing trees is well suited to our scenario: sensor network query processors impose communication workloads on the multi-hop communication network that are very different from traditional ad-hoc networks with mobile nodes. Since the sensor network is programmed only through queries, there are very regular communication patterns, mainly consisting of the collection of sensor readings from a region at a single node or the base station. Note that other types of routing structures beyond routing trees are necessary if the query workload has more than a few destinations since the overlay of several routing trees neglects any sharing between several trees and leads to performance decay. The discussion of such routing algorithms is beyond the scope of this chapter, but we have begun to explore such issues in our research.

3.3 *Query Processing*

Once a query has been disseminated, each node begins processing it. Processing is a simple loop: once per epoch, readings, or *samples* are acquired from sensors corresponding to the fields or *attributes* referenced in the query. This acquisition is done by a special *acquisition operator*. This set of readings, or *tuple*, is routed through the query plan built in the optimization phase. The plan consists of a number of operators that are applied in a fixed order; each operator may pass the tuple on to the next operator, reject it, or combine it with one or more other tuples. Any tuple that successfully passes the plan is transmitted up the routing tree to the node's parent, which may in turn forward the result on or may combine it with its own data or data collected from its other children.

The acquisition operator uses a *catalog* of available attributes to map names referenced in queries into low-level operating system functions that can be invoked to provide their values. This catalog abstraction allows sophisticated users to extend the sensor network with new kinds of sensors, and also provides support for sensors that are accessed via different software interfaces. For example, in the TinyDB system, users can run queries over sensor attributes like light and temperature, but can also query attributes that reflect the state of the device or operating system, such as the free RAM in the dynamic memory allocator. Table 1 shows a list of some of the sensor and system attributes that are available on current generation sensors. This table includes energy per sample as an example of additional catalog metadata that can be used in query optimization.

4 Data Integration Architecture

TinyDB and sensor network query processing systems are one piece of our flexible query processing system for sensor data. However, to make sensor networks truly

Table 1 Some sensors available for Mica motes and their power requirements

Sensor	Notes	Energy per sample (@3V), mJ
Solar Radiation [36]	Amount of radiation in 400 to 700 nm range that allows plants to photosynthesize	0.525
Barometric Pressure [20]	Air pressure, in mbars	0.003
Humidity [35]	Relative humidity	0.5
Passive Infrared [37]	Temperature of an overhead surface	0.0056
Ambient Temp [37]		0.0056
Accelerometer [5]	Measure movement and vibration	0.0048
(Passive) Thermistor [6]	Uncalibrated temperature, low cost	0.00009

useful, users need the ability to seamlessly query data from sensor networks as well as other data sources. Ideally, such a system would provide users with the same query language as well as data layout and schema management tools across both sensors and other data sources. We are in the process of adding support to Borealis (a follow-on to the Aurora [2] stream processing system) to support TinyDB and other limited-capability query processors.

In this section, we consider in broad terms the problem of integrating a sensor database system into any system for processing (non-sensor) data streams (such as Telegraph, Aurora, or other systems described in this book). We note that this problem cannot be trivially solved by simply wrapping the data that is output from TinyDB into statements in the appropriate “data definition language” for the streaming system in question, for the following fundamental reasons:

- TinyDB must be parameterized by one or more queries that specify the data that should be extracted from it. Some mechanism is needed to derive the appropriate query to send into the network when the user requests data from the combined stream/sensor database.
- To provide reasonable performance, as many data reducing operators (e.g., filters and projections) as possible should be executed by the sensors, however;
- TinyDB does not support the full set of operators provided by most streaming systems, so not all queries can be executed in TinyDB. Furthermore,
- Some operators (e.g., joins with external tables) require state to be made available to the sensor network to execute the query in the network, which can substantially increase the cost of executing a query in-network.
- Finally, the optimization metric in a sensor network is typically expressed in terms of energy, whereas it is in other streaming database systems, other, more traditional metrics (e.g., latency) are typically used.

Our basic integrated architecture is shown in Fig. 3. To the right of the dashed line is the data stream processor; to the left is our sensor network system. The proxy between these two systems acts as a mediator, translating queries from the streaming system into queries on the sensor network and managing the optimization process between the two systems.

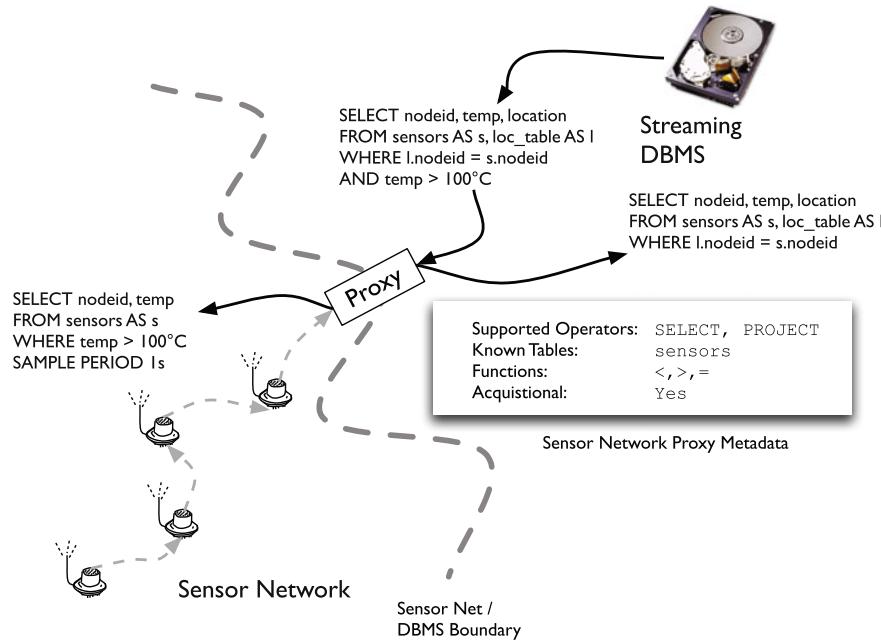


Fig. 3 Proxy architecture

To illustrate how our approach to integration works, we look in detail at how a proxy allows us to solve the two fundamental problems outlined above: query dissemination and optimization.

4.1 Proxy-Based Query Dissemination

The basic process for query dissemination in our architecture works as follows: queries arrive at the data stream management system (DSMS), which parses and verifies them, and sends them to the proxy. The proxy is responsible for transforming these user-input queries into two subqueries: one which runs on the sensor network, the *sensor query*, and one which runs on the streaming database, the *stream query*. The output of the sensor query is fed into the stream query such that the output of this composite stream-sensor query is identical to the non-transformed query where all processing is done inside the stream system.

Since the DSMS parses queries into query graphs of operators, the process of transforming a query in the two subqueries can be thought of as partitioning the query graph. In most cases, the actual mechanisms for partitioning queries are straightforward and well understood from prior work in the database community [22, 29]; connected sub-graphs of the query plan can be pushed into the net-

work, and in some cases operators can be commuted to arrive at plans that provide better performance characteristics.

The challenge of building such a proxy system, then, is determining what decompositions are *feasible*; that is, identifying the decompositions that can actually be executed in the sensor network. A simple option would be to hard-code knowledge into the proxy regarding the capabilities of the sensor network—for example, the proxy might know that sub-plans without joins could be pushed into the network. The problem with this approach is that it constrains the proxy to be usable with a single type of external data source; ideally, we would like to be able to reuse this proxy architecture for other purposes—e.g., to allow for pushing the execution of simple query processing operators into a Website that provides selection facilities via a form-based interface.

Our approach for capability-based approach is loosely inspired by work on mediation systems like Garlic [34]. Unlike in our approach, the Garlic middleware asks client databases if they can run a certain part of each query and at what costs. Depending on the answer, Garlic assigns the parts of the queries to the clients, or runs those parts itself. As we discuss in the next sections, our system uses capabilities to locally enumerate and cost distributed plans at the proxy.

Constraint-Based Capability Language

Thus, we are developing a constraint-based capability language for expressing the set of feasible plans that can be pushed into sensor network query processing services that users wish to interface to the streaming database. To add a new service to the proxy, the user registers a description of that system using the capability language. The language consists of a sequence of property-value predicates. *Properties* are derived from a candidate sensor query and refer to simple characteristics of the query, such as `has_aggregates`. A query is considered valid (i.e., legal to run in the sensor network) if all of the predicates evaluate to true. As a simple example, consider the service description below; it specifies that a given query processor registered with the proxy can accept queries without aggregates and with one or fewer joins of low selectivity.

```
{ has_aggregates = false,
  num_joins ≤ 1,
  all_join_selectivity ≤ 1 }
```

Initially, we provide a small number of basic properties similar to those shown in the capability language fragment above. Adding a new property is a simple matter of writing a function that accepts a query and computes the value of that property over the query. Such functions are a part of the stream query processor, as they may need to access query processor statistics or pieces of the optimizer (e.g., to compute join selectivity).

```

ENUMERATESENSORQUERIES( $s, q$ )
 $r \leftarrow \{\}$ 
 $w \leftarrow \{s\}$ 
WHILE  $w \neq \{\}$ 
 $w' \leftarrow \{\}$ 
FOR EACH QUERY  $curq$  IN  $w$ 
     $Ops \leftarrow \{ \text{NEXT OPERATOR IN } Q[\text{CURQ.ID}] \text{ ALONG EACH NON-OVERLAPPING SUBTREE OF } curq \}$ 
    FOR EACH OPERATOR  $op$  IN  $Ops$  WHERE PARENT( $op$ )  $\neq$  NULL
         $curq' \leftarrow curq \oplus \text{PARENT}(op)$ 
        IF ( $curq'$  IS VALID) THEN  $w' \leftarrow w' \cup \{curq'\}$ 
     $r \leftarrow r \cup w$ 
     $w \leftarrow w'$ 

```

Fig. 4 Sensor query enumeration algorithm

Feasible Plan Enumeration

To derive candidate feasible sensor queries from a service description like the one shown above, the stream query processor begins with a query q , which we will represent as a workflow graph (e.g., a tree of joins, as in a relational database system, or a Stream [26]-style or Aurora [2]-style “boxes and arrows” diagram). It transforms this query by pushing selections past joins and other operators and choosing an ordering for joins (and other commutable operators) that it expects will perform well based on available statistics and optimization goals. The DSMS then produces a simple query, s , that contains no selections or joins and simply projects the fields contained in tables that reside in the sensor network from the original query, q . It then enumerates valid prefixes of query of q that can be added to s , where a valid query is determined by the sensor network’s service description. By *prefix* we mean a traversal from one or more leaves of the workflow graph across one or more operators towards an output of that graph.

The pseudocode for our candidate query enumerator is given in Fig. 4. The basic idea is to try adding successively larger and larger prefixes from q to the set of candidate queries, until any further additions result in invalid plans. We define the operator \oplus to mean the addition of a set of operators contained in a prefix to the query.

More formally, `enumerateSensorQueries` constructs a set, r , of valid candidate sensor queries. Initially, r is empty. The algorithm also constructs a set w of queries that initially contains just s . It then begins looping over w . For each query, $curq$ in w , `enumerateSensorQueries` constructs a set of operators, Ops . Ops contains the next operators along each non-overlapping subtree in $curq$. These operators are found by doing a lookup in the query set q . To construct new candidate plans, `enumerateSensorQueries` loops over each operator op in Ops , and constructs a new plan, $curq'$ where the parent operator of op in q has been added into $curq$. If $curq'$ is valid, the algorithm adds it to the new w set w' . Once it has looped over all plans in w , `enumerateSensorQueries` adds all the plans in w to r and replaces the plans in w with w' .

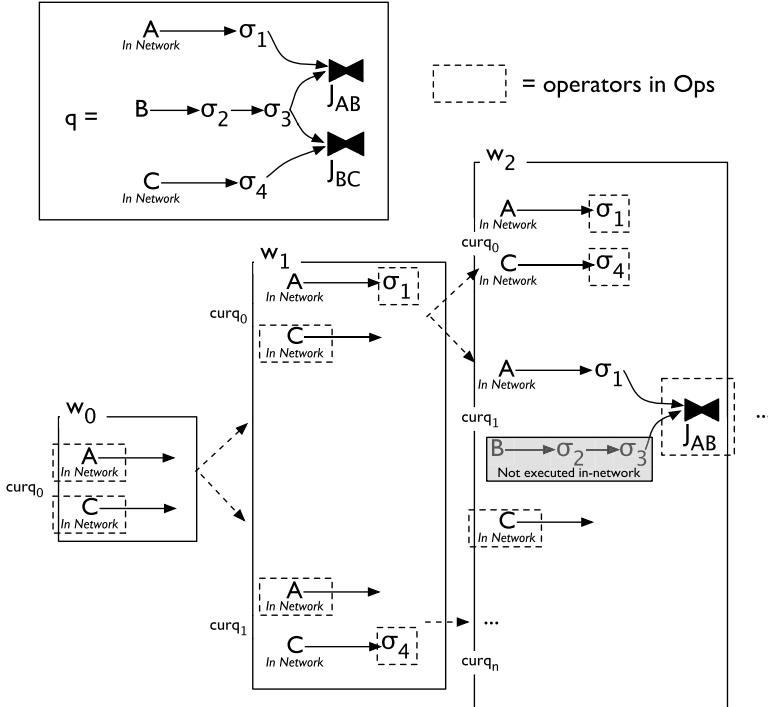


Fig. 5 Illustration of the `enumerateSensorQueries` algorithm. The initial query q is shown, and the contents of w for the first three iterations of the outer loop of the algorithm are illustrated

Note that if we move a join operator into $curq'$, we have joined two subtrees of $curq$, and reduced the number of operators in the set $O_{curq'}$ that results when enumerating the closest operators in $curq'$ versus the number of closest operators in $curq$. Figure 5 shows an example of plan enumeration with the join query over three data sources, two of which originate in the sensor network. The initial plan q is shown at the top of the diagram, and the contents of set w are shown after the first, second, and third iterations of the algorithm. Note that in w_2 , the output of the entire sub-branch of the query containing source B and the two selections over B is pushed into the network. Because the source B exists external to the network, however, we do not ever generate plans that execute operators in B inside the network. Though this is a heuristic (that could be relaxed), we believe it to be a reasonable one as it is unlikely that pushing down sub-plans over external sources will reduce the amount of energy used by the network.² Were B in the sensor network, these operators would, of course, be executed in-network.

²This would only be the case if the cardinality of the output of the sub-branch were larger than the cardinality of the inputs of the sub-branch.

The complexity of `enumerateSensorQueries` scales with the number of distinct in-network sources, S , and the number operators along each path from S to the output of the network. If there are N operators along each such path, the worst case running time will be S^N . Though the running time is exponential, we expect the number of sources in each sensor network to be small, such that the running time will be bounded. Note also that this is a worst-case, as the validity checking will eliminate many possible plans. Simple heuristics that, for example, consider pushing down only a few paths, can be used to bound the complexity of very large problem instances.

4.2 Proxy-Based Query Optimization

Given our set of candidate sensor plans r to evaluate, we can pass them to our query optimizer, which chooses the best sensor query from r , r^* , to push into the sensor network, given the original plan q and the corresponding stream query, \bar{r}^* , that excludes the operators in r^* .

The primary challenge for query optimization across a DSMS-sensor network integration is that a DSMS and a sensor network have different and potentially conflicting optimization goals. The DSMS is responsible for maximizing the QoS (Quality of Service) to end applications, and thus may be interested in decreasing latency, increasing throughput, or maximizing the quality of results delivered to these applications, depending on user preferences. In contrast, a sensor network is primarily concerned with minimizing power consumption in order to extend lifetime and reduce cost. In some cases, these goals conflict. As an example, consider whether a join operation (of sensor data with a static table) should be performed in a sensor network. Assume that the join predicate is highly selective, but that the static table is large so that the table must be horizontally partitioned and the join performed in parallel on many sensors. Assuming the network has the processing capability for the operator, and that the continuous query will run for a large amount of time such that overhead of disseminating the table is outweighed by the join's low selectivity, it is in the sensor network's interest to perform the operation in-network since this will reduce the number of transmitted tuples and thus power used.

To implement such a join in TinyDB, we use a parallel join on nodes within broadcast range of each other so that each sensor tuple is only transmitted once (more details on this approach are given in Sect. 5), with all nodes containing a partition receiving the same transmission. Due to the lossy nature of wireless communication, however, the broadcast might not reach all nodes containing a partition, so some result tuples may be lost, affecting the quality of results observed by the end application. Thus, there is an energy-quality tradeoff: due to power concerns, the sensor network may prefer to perform the join in-network, while, for quality reasons, the DSMS may prefer to perform the join outside the network.

We attempt to resolve this conflict by adding an additional possible optimization metric to our DSMS optimizer—*lifetime*. Lifetime can be thought of as a fixed

amount of energy the DSMS has bought from the sensor network; when this energy runs out, no data will be produced for the query (thus, lifetime is a per-source metric; for now, we define the lifetime of a query to be the minimum of the lifetimes of all data sources in the query). In practice, for many sensor networks, lifetime simply corresponds to the amount of time until the sensor’s batteries are exhausted. The more work the query requires of a sensor network, the faster this fixed amount of energy will be used. Thus, there is an observable cost to the application when making decisions that might improve latency or quality while also increasing power utilization in the sensor network, allowing the DSMS to choose in which direction of the power/latency/quality trade-off to optimize using application QoS functions.

Briefly, query optimization works as follows: after running for some time, the DSMS observes that an optimization dimension, d , is most in need of optimization based on the observed performance of the query and the user’s QoS requirements. For example, if predicted lifetime of the network is 1 week, and the user has requested a lifetime of 2 weeks or more, the optimizer might choose to optimize lifetime. Once the DSMS chooses the optimization dimension, the proxy is notified of this choice. The proxy then searches the set of feasible plans, r , for the plan, r^* that will most improve the desired dimension and sends this plan into the network, returning \bar{r}^* to the DSMS for further optimization and redistribution within the streaming system.

In our current approach, plan costing is done via simple statistics and heuristics as in traditional relational database systems, or as in TinyDB for the case of lifetime [23]. It should be noted that some of the potential optimization dimensions have not been sufficiently studied in the database community. For example, estimating how wireless loss rate of tuples (due to contention) in a sensor network is affected by moving an operator into the network and by different implementations of that operator is complex and potentially difficult to estimate correctly. Since estimation errors are compounded through composition of operators, a more greedy version of the enumeration algorithm described above might be in order, where the search space is pruned by not exploring more than 1–3 operators along any particular branch. This also has the benefit of decreasing the running time of the enumeration algorithm.

If the proxy is already running the maximally efficient plan for the dimension in question, it can attempt to adjust the sensor sample rates to improve d . For example, if d is lifetime or latency, reducing the sample rate will likely improve these metrics. However, if d is tuple quality or throughput, increasing the sample rate might help. If the DSMS indicates that the sample rate should not be changed, the proxy simply notifies the DSMS optimizer that it can do nothing to improve d . We note that this process of re-optimization is fairly expensive and envision it being done rarely. We are investigating lighter weight, local optimizations that can be used to adapt on shorter time scales [1].

5 Join Operator Push down

One of the features that our integrated architecture provides is the ability to push external tables into the sensor network. In this section, we use an example scenario to illustrate an example when this type of push down processing might be useful, and look at the mechanics of executing such joins. The algorithms described in this section are derived from the REED (Robust, Efficient, Event Detection) project, and they are presented in more detail in [4].

Consider the following sensor network query designed for use in a factory environment with temperature sensitive construction phases. The factory produces a product whose creation requires the temperature to remain in a small, fixed range that varies over time. Should the temperature fall outside this range, the product is in danger of being damaged and action must be taken immediately. A continuous query is thus desired that joins aggregate temperature readings from sensors located at various positions in the factory with a time-indexed relation that encodes the desired temperature range. Should the temperature ever fall outside the required range, an appropriate reaction will be initiated.

It might be beneficial to perform a join of this type in the sensor network. First, since the join predicate is highly selective (it only outputs data if something has gone wrong in the temperature maintenance system at the factory), it reduces the number of transmissions that sensors in the network need to make, extending network lifetime. Second, the join can be done in parallel on multiple sensor nodes. The historical table can be horizontally partitioned across sensor nodes and each new tuple joined with each partition separately. This could reduce latency.

The role of the proxy in pushing down the join operator is two-fold. Firstly, it decides whether or not it is beneficial to the application to push down the operator at all. This is handled by the DSMS optimizer described above. Secondly, if the decision is made to move the join into the network, the proxy aids in the distribution of the tuples of the partitioned table to the appropriate sensor nodes.

Once the decision is made to push the join operator into the sensor network, the proxy orchestrates the distribution of the stored tuples to the appropriate nodes. The first step in this process is to flood the description of the operator (including the schema of the input and output tuples) and the name, size, and schema of the stored table down the TinyDB routing tree. Upon receiving this message from the proxy, every node in the sensor network knows whether or not it will participate in the join (by verifying that it contains the sensors that produce the fields in the input schema) and how many tuples of the join table can be locally stored (by calculating the tuple size of the stored table using the table schema and comparing this value with the storage capacity that node is willing to allocate for the query).

Upon receiving this message from the proxy, the sensor nodes then independently form groups. A group is defined as a set of nodes which are all within broadcast range of each other and which collectively have enough storage capacity to accommodate the stored table. The advantage of storing the table in these types of groups is that it takes just one broadcast tuple from any member in the group to reach each location of the partitioned join table. Group formation is a background

task that happens continuously throughout the lifetime of the join. Sensor nodes that produce data relevant to the join but are not currently members of a group initiate a group creation algorithm by notifying neighbors that it is creating a group and using neighbor lists from each willing participant to create a group of maximum size. The initiator then notifies each chosen participant of their acceptance into the group.

Each participant then sends a message to the proxy containing its group identifier and storage capacity. This allows the proxy to decide how to partition the table for this particular group and sends the relevant tuples to each participant according to this decision. Once all tuples have been sent to all participants, the group is ready to perform the join. Since the proxy is aware of what groups have been formed and which tuples have been sent to which nodes, it can deduce where to send updates to the table, should this be necessary.

In summary, with the help of the proxy, a join of sensor data with a static table can be pushed from a DSMS into the sensor network. The proxy administers the distribution of the query and static table into the network, and can monitor and collect query statistics to ensure that the join selectivity remains low enough that executing the join in the network remains the right thing to do [3, 4].

6 Conclusions

Thus, TinyDB provides a robust platform for deploying a wide range of monitoring and data collection applications on sensor networks. By itself, it provides a number of features specially tailored to sensor networks, including features to allow users to control when and how frequently data is acquired from the sensors. When integrated with a DSMS, TinyDB uses a proxy interface to facilitate query optimization between the DSMS and itself, enabling sensor nodes to participate in the processing of complex queries, such as joins between the network and external tables stored in the DSMS.

We believe this integrated data processing architecture is the key to widespread adoption of sensor networks. The vast majority of users are not concerned with the details of power management or ad-hoc, multihop networking—rather, they simply want to combine and process sensor network data with existing data streams and relations, using familiar data processing operations and tools, which is exactly what TinyDB provides.

References

1. D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the Borealis stream processing engine, in *Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA (2005)
2. D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S.Z. Aurora, A new model and architecture for data stream management. VLDB J. (2003)

3. D.J. Abadi, W. Lindner, S. Madden, J. Schuler, An integration framework for sensor networks and data stream management systems, in VLDB (2004), pp. 1361–1364
4. D.J. Abadi, S. Madden, W. Lindner, Reed: robust, efficient filtering and event detection in sensor networks, in *Proceedings of VLDB* (2005), pp. 769–780
5. Analog Devices, Inc. ADXL202E: low-cost 2 g dual-axis accelerometer. <http://products.analog.com/products/info.asp?product=ADXL202>
6. Atmel corporation. Atmel ATMega 128 Microcontroller Datasheet. <http://www.atmel.com/atmel/acrobat/doc2467.pdf>
7. T. Brooke, J. Burrell, From ethnography to design in a vineyard, in *Proceedings of the Design User Experiences (DUX) Conference* (2003). Case Study
8. A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, J. Zhao, Habitat monitoring: application driver for wireless communications technology, in *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean* (2001)
9. J.-H. Chang, L. Tassiulas, Energy conserving routing in wireless ad-hoc networks, in *Proceedings of the 2000 IEEE Computer and Communications Societies Conference on Computer Communications (INFOCOM-00)* (IEEE Comput. Soc., Los Alamitos, 2000), pp. 22–31
10. C Corporation. CC1000 Single Chip Very Low Power RF Transceiver Datasheet. <http://www.chipcon.com>
11. X. Digital Equipment Corporation, Intel. The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 2.0) (1982)
12. J. Elson, L. Girod, D. Estrin, Fine-grained network time synchronization using reference broadcasts, in *OSDI* (2002)
13. D. Estrin, L. Girod, G. Pottie, M. Srivastava, Instrumenting the world with wireless sensor networks, in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)* Salt Lake City, Utah (2001)
14. D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, S. Wickera, Complex behavior at scale: an experimental study of low-power wireless sensor networks. Under submission (2002). Available at <http://lecs.cs.ucla.edu/~deepak/PAPERS/empirical.pdf>
15. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, The nesC language: a holistic approach to network embedded systems, in *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)* (2003)
16. J. Gehrke, F. Korn, D. Srivastava, On computing correlated aggregates over continual data streams, in *Proceedings of the ACM SIGMOD Conference on Management of Data*, Santa Barbara, CA (2001)
17. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D.C.K. Pister, System architecture directions for networked sensors, in *ASPLOS* (2000)
18. D. Inc. company web site. <http://www.dust-inc.com>
19. C. Intanagonwiwat, R. Govindan, D. Estrin, Directed diffusion: a scalable and robust communication paradigm for sensor networks, in *MobiCOM*, Boston, MA (2000)
20. Intersema, Ms5534a barometer module. Technical report (2002). <http://www.intersema.com/pro/module/file/da5534.pdf>
21. D.B. Johnson, D.A. Maltz, Dynamic source routing in ad hoc wireless networks, in *Mobile Computing*, ed. by T. Imielinski, H. Korth. The Kluwer International Series in Engineering and Computer Science, vol. 353 (Kluwer Academic, Norwell, 1996)
22. A.Y. Levy, I.S. Mumick, Y. Sagiv, Query optimization by predicate move-around, in *Proceedings of VLDB* (1994)
23. S. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, The design of an acquisitional query processor for sensor networks, in *ACM SIGMOD* (2003)
24. S. Madden, W. Hong, J.M. Hellerstein, M. Franklin, TinyDB web page. <http://telegraph.cs.berkeley.edu/tinydb>
25. A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, Wireless sensor networks for habitat monitoring, in *ACM Workshop on Sensor Networks and Applications* (2002)
26. R. Motwani, J. Window, A. Arasu, B. Babcock, S. Babu, M. Data, C. Olston, J. Rosenstein, R. Varma, Query processing, approximation and resource management in a data stream man-

- agement system, in *CIDR* (2003)
- 27. V. Park, S. Corson, Temporally-ordered routing algorithm (tora) version 1 functional specification (1999). Internet draft, <http://www.ietf.org/internet-drafts/draft-ietf-manet-tora-spec-02.txt>
 - 28. C.E. Perkins, Ad hoc on demand distance vector (aodv) routing (1999). Internet draft, <http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-04.txt>
 - 29. H. Pirahesh, J.M. Hellerstein, W. Hasan, Extensible/rule based query rewrite optimization in starburst, in *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (ACM, New York, 1992), pp. 39–48
 - 30. G. Pottie, W. Kaiser, Wireless integrated network sensors. *Commun. ACM* **43**(5), 51–58 (2000)
 - 31. G.J. Pottie, W.J. Kaiser, Embedding the Internet: wireless integrated network sensors. *Commun. ACM* **43**(5), 51 (2000)
 - 32. N.B. Priyantha, A. Chakraborty, H. Balakrishnan, The cricket location-support system, in *MOBICOM* (2000)
 - 33. R.F.M. Corporation, RFM TR1000 datasheet. <http://www.rfm.com/products/data/tr1000.pdf>
 - 34. M.T. Roth, P.M. Schwarz, Don't scrap it, wrap it! A wrapper architecture for legacy data sources, in *Proceedings of 23rd International Conference on Very Large Data Bases*, August 25–29, 1997, Athens, Greece (1997), pp. 266–275
 - 35. Sensirion, Sht11/15 relative humidity sensor. Technical report (2002). http://www.sensirion.com/en/pdf/Datasheet_SHT1x_SHT7x_0206.pdf
 - 36. T.A.O. Solutions, Tsl2550 ambient light sensor. Technical report (2002). <http://www.taosinc.com/pdf/tsl2550-E39.pdf>
 - 37. M.M.I. Systems, Mlx90601 infrared thermopile module. Technical report (2002). <http://www.melexis.com/prodfiles/mlx90601.pdf>
 - 38. K. Whitehouse, The design of calamari: an ad-hoc localization system for sensor networks. Master's thesis, University of California at Berkeley (2002)
 - 39. A. Woo, D. Culler, A transmission control scheme for media access in sensor networks, in *ACM Mobicom* (2001)
 - 40. A. Woo, T. Tong, D. Culler, Taming the underlying challenges for reliable multihop routing in sensor networks, in *ACM Sensys*, Los Angeles, California (2003)
 - 41. Y. Yao, J. Gehrke, Query processing in sensor networks, in *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)* (2003)
 - 42. W. Ye, J. Heidemann, D. Estrin, An energy-efficient MAC protocol for wireless sensor networks, in *Proceedings of the IEEE Infocom* (2002), pp. 1567–1576

Part V

Applications

Stream Processing Techniques for Network Management

Charles D. Cranor, Theodore Johnson, and Oliver Spatscheck

1 Introduction

The phenomenal growth of the Internet has had a tremendous effect on the way people lead their lives. As the Internet becomes more and more ubiquitous, it plays an increasingly critical role in society. In addition to leisure-time activities such as gaming and Web browsing, the Internet also carries important financial transactions and other types of business communications. Clearly, our dependency on the correct operation and good performance of the Internet is increasing and will continue to do so.

For network operators, understanding the types and volumes of traffic carried on the Internet is fundamental to maintaining its stability, reliability, security, and performance. Having efficient and comprehensive network monitoring systems is the key to achieving this understanding. The process of network monitoring varies in complexity from simple long term collection of link utilization statistics to complicated ad-hoc upper-layer protocol analysis for detecting network intrusions, tuning network performance, and debugging protocols.

Unfortunately, rapid Internet growth has not made monitoring the network any easier. In fact, three trends associated with this growth present a significant challenge to network operators and the network monitoring tools they use:

- First, Internet applications have become more sophisticated. New applications level protocols with important semantic information in their headers are being layered on top of TCP and UDP. These headers might be free-form text that must be parsed, or the header might implement an application-specific protocol that must be emulated. Network monitoring systems that collect only basic

C.D. Cranor · T. Johnson (✉) · O. Spatscheck
AT&T Labs—Research, Florham Park, NJ, USA
e-mail: johnson@research.att.com

TCP/IP header information are no longer sufficient for evaluating and debugging application-level performance. Additionally, these new protocols are placing stricter demands on Internet performance—in terms of bandwidth, latency and loss.

- Second, ever increasing Internet data rates result in more data traversing the network making it harder to examine specific parts of network data in any detail.
- Third, Internet users are becoming more savvy and are more concerned with application-level performance than with network-level statistics that network operators typically collect and report.

We found that the current generation of network monitoring tools such as SNMP [6], RMON [24], NetFlow [7], and tcpdump [13] no longer fully address network monitoring and debugging needs. A serious problem is that the trends make not only monitoring, but also data management difficult. Current generation tools create very large collections of flat files, which are analyzed with hand-crafted programs (for example, Perl scripts). Managing these very large data sets is difficult—data quality problems are common and metadata is quickly lost. The data analysis often involves merging data sets and correlating data from multiple sources (e.g., correlating TCP performance data with BGP—Border Gateway Protocol, a router update protocol [20]—data). For alerting and triggering applications, we need to evaluate complex queries in real time.

To address these problems, we have created Gigoscope—a fast and flexible stream database for network monitoring. Gigoscope was designed around two key aspects. First, Gigoscope has a highly flexible SQL-like query language, GSQL, for its interface. Using a database query language provides us with great flexibility and allows Gigoscope to be quickly adapted for new problems—only the high-level query need be changed. For example, Gigoscope can be used to analyze or monitor newly evolving peer-to-peer and multi-media protocols (which change often). Network operators are especially interested in these types of protocols because of their potential impact on traffic patterns.

Second, Gigoscope was designed using the overriding principle of reducing data as early as possible to allow high-speed monitoring. Gigoscope queries are automatically broken up into hierarchical components. Low-level components can run on the network interface card itself, reducing data before it reaches the main system bus. High-level query components may run either in kernel or user space and can be used to extract application layer information from the network. By reducing data in key locations, a single machine can monitor very high-speed links. Early data reduction also makes it practical to use high-level languages such as Perl to interpret the results of Gigoscope queries.

In this chapter, we discuss stream processing techniques for network management, especially as implemented using Gigoscope. We discuss the nature of network performance and management data and the types of queries that are typically posed. Next, we discuss the GSQL language, and show how it is designed to support queries on network data. Network data streams can be very high volume (multiple Gbits/s and millions of records per second) and require a highly optimized processing archi-

ture, which we discuss next. We present some observations about performance, and conclude with discussions of some sample applications.

2 Network Monitoring

Network operators perform network monitoring for a variety of reasons, ranging from routing optimization to application performance monitoring. While these widely disparate activities have their unique characteristics, there are also many commonalities. In this section, we discuss the nature of the data and queries used for network monitoring.

2.1 Network Monitoring Data

Network monitoring data generally arrive as a data stream—the latency between the observed activity and the arrival of its measurement is small, and the data arrives continually as it is highly undesirable to turn off the network.

The measurement data can be archived to disk and later loaded into a conventional database or analyzed using ad-hoc tools. However, there are severe problems with these approaches. Even moderate speed networks generate huge amounts of data. For example, monitoring a single gigabit ethernet running at 50 % utilization generates a terabyte of data in less than five hours. Most of the data is not valuable for (legitimate) analyses. This approach therefore presents intractable problems of cost, speed, security, and privacy. The speed issue is critical because for many applications the value of the result degrades rapidly with time (you want to trigger an alarm seconds after a DDOS attack starts, not days after). Stream processing techniques which reduce these huge volumes of data into actionable knowledge in real time are critical for network monitoring.

Current generation monitoring tools such as SNMP and Netflow provide a pre-aggregated view of network activity. SNMP typically maintains a collection of counters (e.g., of bytes or packets traversing a network interface) and reports the value of these counters at regular intervals (e.g., once a minute). Netflow data is a byproduct of a router’s “flow” cache. A *flow* is a sequence of packets from a particular source to a particular destination such that the time between successive packets is small. When the flow finishes, the router emits a flow report specifying the source, destination, byte and packet count, start and end time, and some router-specific information. The router does not monitor each flow to determine when the flow ends, instead flows are generated as a byproduct of the flow cache cleaning algorithm (long-lived flows are also emitted for timely monitoring). The precise details of flow generation depend on the router’s operating system and configuration parameters.

SNMP and Netflow are effectively the output of simple aggregation queries which reduce the volume of monitoring data to manageable levels. Their volume

can still be large, e.g., Netflow monitoring data from core routers, so further reduction in a stream database is often necessary. We found that Gigascope queries are a better mechanism for this kind of pre-aggregation, being more flexible and with more precisely defined semantics. However, it is often cheaper and easier to use Netflow for very large scale monitoring because Netflow is produced by the router,

Netflow and SNMP summaries are often too coarse for the intended analysis. For a more detailed view, we need to look at packet data. Many analyses are made from packet headers—the data volume is reduced by projecting out the packet body. Unfortunately, a great deal of information about the application protocol (web, multimedia, P2P, etc.) is thereby lost. Using the port number to deduce the protocol is unreliable because of port 80 tunneling (e.g., to get through firewalls) and the use of non-standard port numbers (e.g., to get around P2P port blocking). With a stream database such as Gigascope, we can access the packet body in an efficient way and recover application protocol information.

Our primary sources of network performance data are either packet data streams or packet aggregate data streams. However, other performance data streams are available. For example, we have incorporated active network measurements into our Gigascope queries by making them into data stream. Some particular characteristics of these data are:

- The data arrives as a stream.
- Each tuple in the stream contains a field which acts as a timestamp or a sequence number (e.g., the end time field of a Netflow, the arrival time of a network packet, the TCP sequence number, etc.).
- A tuple might have several timestamps, which have different properties. For example, a Netflow tuple has both a start time and an end time. The stream is likely to be sorted on the end time, and almost sorted on the start time (it will be within a window, e.g., five minutes, before the end time).

The network performance data does not have much meaning in isolation. Another important type of data is configuration data. For example, to interpret network performance data we often need to understand the state of the routers. One data feed is updates of router configuration data, typically provided as daily snapshots. Another feed is a stream of route update messages, for example, messages exchanged by BGP. Analyzing BGP and related streams is also of independent interest.

2.2 Network Monitoring Queries

Queries over network data depend on the application. The archetypical network query is load monitoring, usually long term and for the purpose of network optimization. This application is a popular example because it can be answered using coarse grained summaries such as Netflow or SNMP data. However, a flexible and high speed tool such as Gigascope enables a wide range of critical applications.

Fine Grained Load Monitoring

Provisioning network links to properly support customer applications requires good understanding of the types of loads currently in the network. The network operator needs to ensure the good performance of sensitive applications while providing adequate capacity for all applications. The desired report is usually an aggregation of traffic volume grouped by time period and application type (determined from the port number). Traditionally, this monitoring is done with a combination of SNMP and NetFlow statistics. Unfortunately, the bandwidth measurements these tools make uses a fixed level of aggregation that can result in undesirable smoothing of traffic characteristics. Gigascopes can be used to mitigate this problem because they can easily be programmed to provide very fine-grained load information. Tools such as NetFlow are difficult to change as they are embedded within a router's firmware.

Hidden Traffic Detection

The advent of peer-to-peer networks has created a management challenge for network operators because it soaks up lots of bandwidth and contains data that may be illegal to copy in the first place. Because of the impact of peer-to-peer traffic, it is important for network operators to keep track of how much load it is placing on network links. However, the traditional approach of monitoring well known TCP and UDP port numbers in NetFlow records to track a class of application's network usage breaks down in peer-to-peer networks because peer-to-peer protocols have been designed to use random or misleading port numbers in order to circumvent firewalls and rate-limiting routers. A tool such as Gigascope can detect hidden traffic by filtering application-level protocol headers in the data area of TCP/IP packets. This type of filter usually involves searching for particular combinations of keywords. These protocols evolve, but it is easy to update Gigascope queries to track the new versions.

Customized Application Monitoring

Network operators are often be requested to examine the performance of a critical customer application and provide some application-layer performance metrics. This type of analysis can be difficult to do because it is often necessary for the network monitor to understand the application-level protocol. Furthermore, the most meaningful statistics are generated by emulating the protocol (perhaps TCP) and reporting, e.g., lost packets and round trip times. We often need to provide complex statistics, such as distributions (i.e., quantiles) of round trip times. Identifying the source of performance problems sometimes requires correlating this performance data against active network measurements, router configuration data, and route update protocol data.

Network Attack Detection

Networks are under frequent attack, necessitating rapid responses. Attack detection involves looking for “anomalous” behavior (sudden increase in traffic) and/or for known attack signatures (port scanning or detecting a known worm signature). When an attack is discovered, the appropriate action is to generate an alert and to look more closely at the attack traffic.

3 Gigascope

We developed Gigascope to address the monitoring and data management shortcomings of current generation monitoring tools. Gigascope is a high-performance stream database specialized for network monitoring. To ensure our ability to achieve high performance, Gigascope has some simplifying restrictions. The primary restriction is that Gigascope is a pure stream database, and does not even directly support conventional relations much less continuous query tables. We have used Gigascope to develop a diverse collection of network monitoring applications and have not found these restrictions to be serious.

3.1 Query Language

The Gigascope query language, *GSQL*, is a pure stream query language with SQL-like syntax (being mostly a restriction of SQL). That is, all inputs to a GSQL are streams, and the output is a data stream. We feel that this choice (akin to that made by Tribeca [21] and Hancock [9]) allows for precise query semantics, enables the composition of complex query processing, and simplifies the implementation of fast operators.

Almost all network data contains one or more timestamps or sequence numbers, and almost all monitoring queries make explicit reference to one or more of these timestamps. We use these timestamps to explicitly limit the locality of the output tuples that an input tuple contributes to. All of our operators, including aggregation, join, and merge, are automatically unblocking and are implemented using simple modifications to conventional pipelined implementations. There is a cost, however, as GSQL cannot express monitoring queries which require the continuous query model.

Ordered Attributes

One concern in a stream database language is that of *blocking operators*. While some operators (such as selection and projection) need no state other than the tuple

being processed, other operators (such as aggregation and join) potentially require their entire inputs before a single output can be produced. One approach to bounding the state is to use sliding windows [3]. However, our approach is to analyze the “timestamps” of the input stream(s) and the properties of the query to determine a query plan which bounds the state required to evaluate blocking operators.

We observed that network analysis data generally contains one or more timestamps or sequence numbers, that these timestamps generally increase (or decrease) with the ordinal position of a tuple in a stream, and that almost all queries make reference to these timestamps. We therefore adopt an approach similar to that of a sequence database [19]. However, a sequence database model has a couple of limitations which make it impractical for our application. First, network data streams often have several timestamps and sequence numbers, and they might not be monotonically increasing with the ordinal position of the tuple in the stream. For a simple example, Netflow records have a *start* and an *end* timestamp. A stream of Netflow records produced by a router will have monotonically increasing *end* timestamps, and generally (but not monotonically) increasing *start* timestamps. Further, most queries on Netflow data will refer to the *start* timestamp rather than the *end* timestamp. The notion of sequence is further perturbed by operators such as join and aggregation (e.g., Netflow is the result of an aggregation query). Second, network analysis queries naturally involve predicates and other references to the timestamps and sequence numbers, but not to the ordinal position of a tuple in its stream.

We make use of timestamps and sequence numbers by defining them to be *ordered attributes* having *ordering properties*. These properties might be inherent in the data source, or might be due to processing by an operator. Below is a illustrative but nonexhaustive set of ordering properties:

1. **Strictly/monotonically increasing/decreasing** expresses the usual notion of a timestamp.
2. **Monotone nonrepeating** is a generalization of monotone increasing, and might occur due to a hash function.
3. **Banded-increasing(ϵ)** is a modifier of the increasing property, and states that the attribute will always be less than ϵ below the high water mark. The start time of a Netflow record is banded-increasing (5 minutes) if all Netflows are dumped by the router every 5 minutes. This property can also occur in the output of a windowed join.
4. **Increasing with probability p** is another modifier of the increasing property, and states that there are occasional exceptions (at rate p) to the ordering property.
5. **Increasing in group G** states that among the tuples defined by the field in G , the attribute is increasing. This property can occur after aggregation. For example, the *start* time of a Netflow record (an aggregation of packets) is increasing in group $\{sourceIP, destIP, sourcePort, destPort, protocol\}$.

We might need to modify these definitions to account for almost-sorted input. For example, Netflow records are sorted on the end time, and all Netflow records are dumped every 5 minutes. Therefore, the start time of a record is always within 5 minutes of the high water mark, i.e., the *start* attribute is banded-increasing (5 minutes).

We use the ordering attributes to turn blocking operators into stream operators. In the current implementation of Gigascope, we use the monotone increasing property as follows:

- **Join.** The join predicate must contain a constraint on an ordered attribute from each table which can be used to define a join window. For example, $B.ts = C.ts$ or $B.ts \geq C.ts - 1$ and $B.ts \leq C.ts + 1$.
- **Group-by and aggregation.** The group key must contain at least one ordered attribute. When a tuple arrives for aggregation whose ordered attribute is larger than that in any current group, we can deduce that all of the current groups are *closed* and will receive no further updates in the future. All of the closed groups are flushed to the output.

The Gigascope data definition language allows the user to specify special properties of the attributes in a source stream, including the ordering properties. The query processing system will impute ordering properties of the output of query operators. For example, suppose that attribute ts is monotonically increasing and a projection operator computes the value $ts/5$ as one of the attributes in its output. We can impute that $ts/5$ is also monotonically increasing. We can perform similar reasoning for the group-by/aggregation operator.

The ordering property imputation for the join operator is more complex, and depends on the constraints in the join predicate and the particular join algorithm selected. For example, if $B.ts$ and $C.ts$ are monotonically increasing, $B.ts$ is in the output, and the join predicate contains the constraint $B.ts = C.ts$, then $B.ts$ in the output will be monotonically increasing. If the constraint is $B.ts \geq C.ts - 1$ and $B.ts \leq C.ts + 1$, then in the output $B.ts$ might be monotonically increasing or banded-increasing⁽²⁾ depending on the choice of join algorithm (monotonically increasing requires more buffer space).

Our approach is similar to that of punctuation semantics for data streams [23]. By labeling attributes with ordering properties, we make the tuples self-punctuating.

3.2 The GSQL Language

GSQL is an SQL-like stream database language, being mostly a restriction of SQL but with some stream database extensions. Currently GSQL supports selection, join, aggregation, and *stream merge* (discussed below). Join queries are currently restricted to two-stream joins, and the join predicate must include a constraint which defines a window on ordered attributes from both streams. Aggregation queries *should* have at least one ordered attribute as one of the group-by keys, but this restriction is not enforced (the user can obtain output by flushing the query).

All queries operate over streams, which come in two flavors: *Protocols*¹ and *Streams*. A Protocol is a data stream generated by interpreting a sequence of data

¹This word was chosen because of its connotations to the end-users.

packets which are presented to the Gigascope run time system. These data packets can be from any reasonable source—IP packets transported via OC48, Netflow packets, BGP updates, etc. The Gigascope run time system interprets the data packets as a collection of fields using a library of interpretation functions. The schema of a Protocol stream maps field names to the interpretation functions to invoke. A Stream is the output of Gigascope query. The fields of its tuples are packed in a standard fashion.

A Protocol defines a mechanism for interpreting a data source, but not what serves as the data source (whereas the source of a Stream is the output of a query). To completely specify a data source, the Protocol must be bound to an *Interface*—a symbolic name which the run time system can bind to a source of packets (if no Interface is given, a default Interface is implied). An example which reports the destination IP and port, and a timestamp from TCP packets on eth0 (the first Ethernet interface card) is:

```
DEFINE{ query_name tcpDest0; }
  Select destIP, destPort, time From eth0.TCP
    Where IPVersion = 4 and Protocol = 6
```

The DEFINE section of a query allows the user to set properties of the query. In this case, the query name is set to tcpDest0. A user application or another GSQL query can read the output of tcpDest0 by specifying it in the From clause.

Network data streams often come from many sources, which might be analyzed separately or in combination (for example, packets flowing to and from a client site and the internet). Therefore, GSQL contains an extension to SQL, the *merge* operator, which is a Union operator which preserves the ordering properties of an attribute. For an example of a merge query, suppose that we have a tcpDest1 which matches tcpDest0 except that it reads from Interface eth1:

```
DEFINE{ query_name tcpDest; }
  Merge tcpDest0.time : tcpDest1.time
    From tcpDest0, tcpDest1
```

The merge operator allows us to combine streams from multiple sources into a single stream. This operator is surprisingly important—we implemented it before the join operator. We developed Gigascope to monitor optical links, which are usually simplex rather than duplex. To obtain a full view of the traffic on a logical link, we need to monitor two physical interfaces and merge the resulting streams.

GSQL supports the join of two streams as long as it can determine a join window from the join predicates. However, GSQL does not currently support the join of a stream to a non-stream relation. Instead GSQL provides support for user-written functions which can act as special types of (foreign key) joins. These have worked so well in practice that supporting non-stream tables in GSQL has become a low priority.

Users can make new functions available by adding the code for the function to the function library, and registering the function prototype in the function registry. In the function registry, the function can be marked as a *partial* function, meaning

that it might not return a value. The processing is the same as if there is no result from a join—the tuple being processed is discarded. One or more of the parameters of the function can be marked as *pass by handle*. These parameters (which must be literals or query parameters) require expensive pre-processing before the function can use them, for example, a regular expression to be compiled. Let us consider an example:

```
DEFINE{ query_name tcpByAS; }
  Select peerid, tb, count(*) as Cnt From tcpDest
  Group by time/60 as tb,
    getlpmid(destIP,'peerid.tbl') as peerid
```

The attribute *time* is a 1-second granularity timer, so *time/60* defines minute-long buckets (when group with a new value of *tb* is produced, all of the pre-existing groups are closed, and therefore are flushed to the output stream). The *getlpmid* function performs *longest prefix matching*—that is, it identifies which subnet an IP address belongs to. Longest prefix matching is a common network analysis activity, and researchers have developed special fast algorithms for it, which *getlpmid* implements. The second parameter is a pass-by-handle parameter, which indicates a file mapping each routable prefixes in the Internet to the ISPs which originates the prefix (i.e., obtained from a routing table). Before the *getlpmid* function is first invoked, the parameter handle registration function reads this file and builds a special in-memory data structure for the function. This in-memory data structure referenced by the handle is then used when the *getlpmid* function is called to quickly map IP addresses to the ISP which originates them.

Many network monitoring queries require the emulation of a network protocol. While it is possible to express many protocol emulation subqueries in SQL, usually it is much easier and more efficient to write C or C++ code. To capture these subqueries, Gigascope provides a mechanism for expressing *user-defined operators* as stream operators. The stream properties of the operator are expressed using an *operator view* wrapper. For example, a piece of code which estimates packet loss rate in TCP/IP connections aggregated by autonomous system and 60 second intervals can be exposed to Gigascope using the following wrapper:

```
OPERATOR_VIEW tcp_loss {
  SOURCE(file tcp_loss);
  FIELDS {
    UINT time (increasing); UINT peerid; loss_rate_estimate; }
  SUBQUERIES {
    tcpq (UINT time (increasing), UINT srcIP, UINT destIP,
      UINT srcPort, UINT destPort, UINT seqNum, UINT ackNum );
  SELECTION_PUSHDOWN {
    srcIP -> tcpq.srcIP; destIP -> tcpq.destIP;
    srcPort -> tcpq.srcPort; destPort -> tcpq.destPort; }
}
```

Here we specify that the user-defined operator *tcp_loss* receives as input a data stream named *tcpq* with the specified fields and ordering properties and producing

the specified output. The SELECTION_PUSHDOWN clause provides query transformation hints to the optimizer. In the following example, we join `tcp_loss` output with `tcpByAS` to produce a combined report:

```
DEFINE{ query_name tcp_perf; }
  Select C.peerid, C.tb, C.Cnt, L.loss_rate_estimate
  From tcpByAS C, tcp_loss L
  Where C.peerid = L.peerid and C.tb = L.time
```

Network analyses often make use of complex holistic aggregates, such as quantiles and heavy hitters. In recent years, a significant literature has developed which describes how to approximate complex aggregates quickly and using small space. Gigascope incorporates a mechanism for defining user-defined aggregate functions (UDAFs), which we have used to implement several streaming algorithms. For more details, see [8].

4 Architecture

In this section we describe the Gigascope architecture, starting with an overview, and then examining the query interface, run-time system, and network device interface support in detail.

4.1 Architectural Overview

To build a Gigascope application, you first need an idea of the type of network measurement that needs to be made. Once that has been established, then the query set (usually more than one query is required for the application) must be expressed in the GSQL language (described in detail in Sect. 3.2). The GSQL query set expression is then input into the Gigascope query compiler. The Gigascope query compiler takes a set of GSQL queries as input and generates a set of query nodes, expressed as C and C++ modules.

To run a Gigascope query, the query node modules are compiled into a library and linked into the run-time components of Gigascope. When the top-level Gigascope application is started, it allocates and configures a set of queries required to support the application. The top-level application collects the resulting output streams of its queries, which can be immediately displayed, used to trigger actions, or saved for loading into a conventional data warehouse.

The query nodes (called *FTAs* within Gigascope for historic reasons) execute in an environment provided by the Gigascope run-time system. The run-time system is responsible for managing and tracking all FTAs, handling IPC between Gigascope components, and tracking available network input devices. The network input devices themselves are managed by a device interface layer. This layer provides a

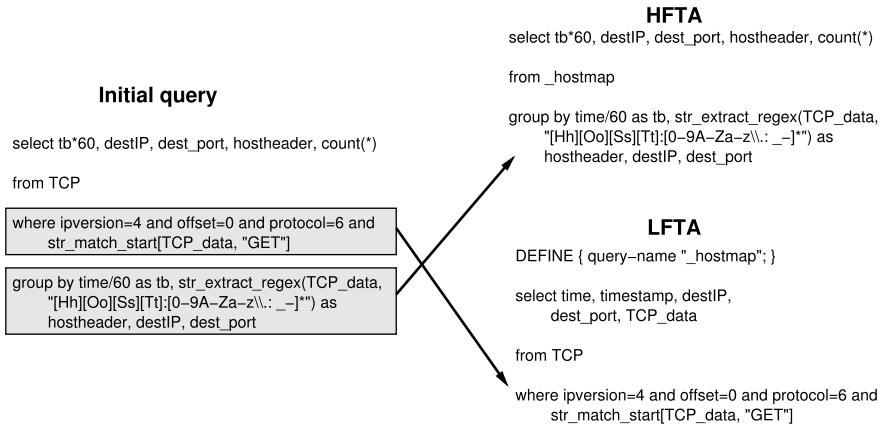


Fig. 1 Example GSQL query and how it is split into an LFTA and a HFTA by the query compiler

uniform interface to the various types of network interface hardware that Gigascope is capable of using, including hardware that uses custom Gigascope-based firmware.

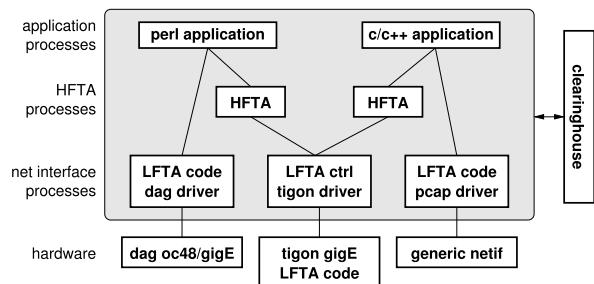
When an FTA is created, it takes a set of configuration parameters as input. This allows a single FTA module to perform different types of queries based on the parameters. Once allocated, the FTA can receive input data streams from the network and other FTA modules, and produces an output data stream.

There are two types of Gigascope FTAs: low-level FTAs and high-level FTAs, called LFTAs and HFTAs, respectively. LFTAs are designed to read data from the device interface layer and may run on the network interface card itself. Thus, LFTAs need to be capable of running in a resource constrained firmware-based environment. On the other hand, HFTAs receive their tuples from LFTAs or other HFTAs and run on the host itself, so they have fewer operational constraints. An example is depicted in Fig. 1.

The practice of splitting queries into high-level HFTAs and low-level LFTAs is crucial for monitoring high-speed packet streams without packet loss. In the best case, we can migrate the LFTAs to the network interface card and avoid the need to move most packet data to the host. Even when the LFTAs execute on the host, this architecture provides significant benefits. The largest buffers in the system are those for the network streams. The lightweight and fast LFTAs process the network streams, creating vastly reduced streams for slower and more extensive processing by the HFTAs. In this way, we minimize memory buffer requirements and therefore packet loss rate.

We note that the FTA modules and the output that they generate are self-describing. This allows applications to query the module or the module output directly in order to determine and decode its format. This feature prevents query output from getting out of sync with the query that generated it in the event that the query was modified.

Fig. 2 Gigascope run-time architecture



4.2 Gigascope Run-Time System

The Gigascope run-time system provides the software and hardware environment that Gigascope queries run within. The run-time systems is responsible for:

- Providing a high-level application interface. This API is used to allocate and configure FTAs, start and stop query processing, and collect query output tuples.
- Keeping statistics on the overall system operation. These statistics include information on system load and are used to fine tune query processing.
- Defining the processing environment that FTAs run in. This includes determining how allocated HFTAs and LFTAs are assigned to processes and coordinating IPC and shared memory management among those processes. The run-time system keeps a registry of all FTAs and processes associated with a running Gigascope.
- Providing GSQL's packet data access and external functions and UDAFs. References to these functions are generated by the query compiler.
- Providing a uniform network device interface to FTA query code. This allows FTA module code to be independent of the underlying network interface being used. Gigascope currently supports three network interfaces: Libpcap, Tigon, and DAG. Libpcap is a portable low-speed interface, Tigon is a high-speed firmware-based gigabit ethernet interface, and DAG is a high-speed gigabit ethernet and OC48 interface.

Figure 2 shows the run-time architecture provided by the Gigascope run-time system. The main run-time system process that tracks allocated FTAs and manages IPC is called the “clearinghouse” process. When a user application process starts, it contacts the clearinghouse process in order to establish a communication path with the Gigascope run-time system. It then uses the run-time system’s high-level application interface to start a query. This involves allocating HFTAs and LFTAs for the query and arranging for their output tuples to be directed back to the application for processing. Some queries do not require the use of HFTAs—in this case the user application process receives output tuples directly from LFTAs.

User applications can be written in a variety of languages including C/C++ and Perl. Interpreted applications are acceptable even in a high-speed environment provided that Gigascope can aggregate the input data down to a manageable size before handing the output tuples off to the interpreter. The Perl binding for the Gigascope

application API returns tuples as associative arrays making it easy to access and process query data.

The Gigascope run-time system groups all LFTAs into a single process, while it assigns HFTAs to a set of processes. Depending on the network device interface, the LFTA process may run on the host system or on the network device itself as part of custom Gigascope firmware. Running the LFTAs on the network device allows Gigascope to improve performance by significantly reducing input data bandwidth before it even hits the system bus. Even without special hardware, running LFTAs in a single Unix process works well because the data is reduced before it is distributed through the Gigascope IPC system to the HFTA processes. Furthermore, on the dual processor systems we typically use for Gigascope, it is safe to assume that one processor can be devoted to performance critical LFTA processing while the other can be shared among the HFTAs and user applications.

4.3 Gigascope Network Device Interface

In the previous section, we introduced the three network devices that Gigascope currently supports: Libpcap, Tigon, and DAG. In this section, we examine Gigascope's device support in more detail. Of the three device interfaces, the Libpcap interface is the simplest as no special hardware support is required for it. Although Libpcap is a relatively low-performance interface, the main advantage of it is that it is portable across many systems. Libpcap allows Gigascope to run on any system that can run the Tcpdump application. For the remainder of this section, we will focus on the operation of the more complex hardware-based DAG interface (we do not have space to discuss the Tigon interface).

DAG Network Interface

The DAG is a PCI network monitoring card manufactured by Endace that is capable of monitoring very high speed networks. We are currently using both gigabit ethernet and OC48 DAG cards with Gigascope. Each DAG card contains a packet processor, a radio clock interface, a network MAC interface, and a high-performance PCI bus interface. Unlike the Tigon, the DAG card uses host memory to store packet data. The DAG clock interface allows the DAG to provide very precise timestamps with captured packets. Details on the DAG hardware can be found on the Endace Web site [11].

Gigascope's DAG software environment consists of two main parts provided by Endace: a device driver used to control the card, and a small library used by applications such as Gigascope to access packets captured by the card. When the host system boots, the DAG device driver allocates a large physically contiguous buffer in host memory for use as a receive ring buffer. This ring buffer is directly mapped

into the LFTAs address space. Gigascope polls this ring buffer to detect when new packet data arrives and should be processed by the FTA code.

Like Tigon-based Gigascopes, the DAG run-time architecture requires special device driver support and uses polling to avoid interrupt overheads. However, in the DAG case LFTAs run on the host rather than in the card. Network data is delivered directly to the application LFTA process, completely bypassing the kernel, but not the system PCI bus. Some Endace cards can perform filtering to reduce the data load. Gigascope models these filters as selection and projection operators.

5 Example Applications

We conclude our evaluation with some real-world measurements from deployed Gigascopes. For business reasons, we have anonymized the data presented here.

Fine-Grained Load Monitoring

We deployed a Gigascope to measure network traffic in parallel with an existing NetFlow-based measurement system on a link in an internet data center. For the Gigascope, we used an aggregation interval of 1 second. The NetFlow aggregation interval is at least 30 seconds, but can be longer depending on the behavior of the flow. We measured the same traffic for a period of one day with both Gigascope and NetFlow. Averaged over the full day, both Gigascope and NetFlow measured 2.1 kbps. However, the finer grained Gigascope measurement found peak bandwidths up to 729 kbps, while NetFlow only reported peaks of 190 kbps indicating that the higher peaks were getting reduced by the relatively coarse grained NetFlow aggregation.

Hidden Network Traffic Detection

We deployed a Gigascope within an access network and collected bandwidth data on peer-to-peer traffic. We collected our data using two queries. The first query used only well-known TCP/UDP port numbers to identify peer-to-peer traffic. This corresponds to the type of measurement that can be made with NetFlow. On the other hand, the second query searches for peer-to-peer traffic by searching packet payloads for application-level peer-to-peer protocol headers. This corresponds to the kind of flexible queries that can easily be made with Gigascope to detect hidden network traffic. One day's worth of results for our queries is shown in Fig. 3. The plot clearly shows that the Gigascope query detects more than two times the amount of peer-to-peer traffic that the simple port-based query detects.

Fig. 3 Hidden network traffic detection

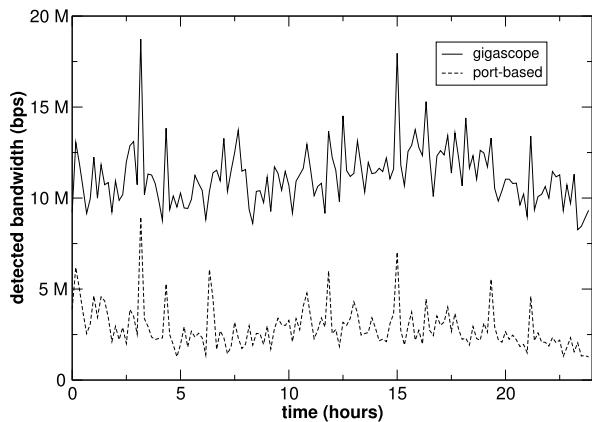
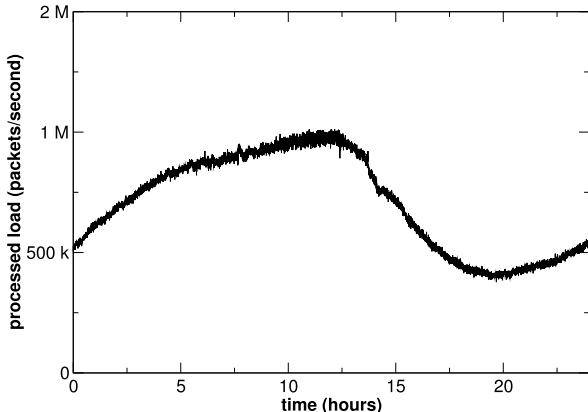


Fig. 4 Customized application monitoring



Customized Application Monitoring

Finally, we deployed Gigascope on a customer link and wrote a custom query to analyze the application-level performance of the customer's protocol. Our query tracks the number and IP address of all application users and provides detailed performance statistics for 5 % of all users. The detailed statistics include round-trip time, client loss, client latency, server loss, and server latency. We also capture all ICMP unreachable error messages to track network errors. The processed load for one day's worth of traffic is shown in Fig. 4. At the peak load of 1M packets/second the Gigascope is monitoring 90K application users generating 1.4 Gbps of traffic.

6 Related Work

One of the most common network monitoring tools is the `tcpdump` [13] program. `Tcpdump` is based on a packet capture library (`libpcap`) that uses the Berkeley packet

filter (BPF) [16] to capture network traffic. Both Gigascope and tcpdump were designed to enable network data to be reduced as early as possible. Tcpdump achieves this goal by providing a simple configuration language that allows the specification of a single BPF filtering expression.

The Windmill [15] framework uses a similar approach as tcpdump to reduce data early on in the kernel. However, in addition to the basic filtering tcpdump performs, Windmill provides a set of protocol modules that allow the reassembly of protocol information.

Another way to reduce data volumes early is to execute trusted code in the kernel. This approach is used by the FLAME [1] architecture. The main drawback of such an approach is that the semantics of the code that is executed in the kernel is only known to the programmer thus making it difficult to automatically optimize the execution of a query.

In addition to research efforts in network monitoring, there are also a large number of commercial products in this area. These products range from simple flow based monitoring tools to complex application-layer analyzers. The most dominant flow based tool is the Netflow [7] tool implemented on a large number of CISCO routers. Netflow operates by counting packets traversing the router hashed on their IP addresses and port numbers. The router will periodically flush its Netflow hash table based on time and current network load.

In the area of application-layer analyzers, the most promising products appear to be the solutions offered by Narus [17] and Nixsun [18]. Both of those solutions provide a set of predefined configurable reports. However, neither provides the flexibility to users to specify new queries which then get executed on the line cards. In addition, our experience with vendors shows that persuading them to add new functionality to their network monitors is expensive and has a long lead time.

From an architectural perspective, Gigascope borrows heavily from the x-kernel [12]. Gigascope takes x-kernel abstractions, applies them to GSQL queries, and implements them using a combination of message queues, shared memory, and firmware.

The database community has developed the concept of a *data stream* [14] which is similar to the concept underlying GSQL. Data streams are used to describe data sets that arrive in a continual fashion, such as financial data, sensor data, and network traffic data. One approach proposed for data stream databases is the *continuous query model* [2, 4, 22].

The database model used for GSQL is most similar to the recent body of work [5, 10, 14] that explores the problem of query evaluation over sensor networks. A significant difference between Gigascope and this work is that Gigascope is targeted towards high performance applications, with a corresponding emphasis on performance optimization.

7 Summary

Our experiences with large scale network analysis convinced us that conventional methods and tools had significant limitations. We developed a stream database, Gigascope, to address these problems. By expressing network sniffing as SQL queries and applying careful optimizations, we developed a very high speed, very flexible, and rapidly reconfigurable network monitor. Furthermore, much of the required data analysis is performed by Gigascope. The output is readily loaded into conventional stored-table databases and queried with SQL, providing a consistent developer's interface.

References

1. K.G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M.B. Greenwald, J.M. Smith, Efficient packet monitoring for network management, in *Proceedings of IFIP/IEEE Network Operations and Management Symposium (NOMS)* (2002)
2. A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom, Characterizing memory requirements for queries over continuous data streams, in *Principles of Database Systems* (2002)
3. B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in *Principles of Database Systems* (2002), pp. 1–16
4. S. Babu, J. Widom, Continuous queries over data streams. *SIGMOD Rec.* **30**(3), 109–120 (2001)
5. P. Bonnet, J. Gehrke, P. Seshadri, Towards sensor database systems, in *2nd Intl. Conf. on Mobile Data Management* (2001)
6. J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin, *RFC 1157: Simple Network Management Protocol (SNMP)* (1990)
7. Cisco. Netflow services and application. <http://www.cisco.com/>
8. G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, D. Srivastava, Holistic udfs at streaming speeds, in *Proc. ACM SIGMOD* (2004)
9. C. Cortes, K. Fisher, D. Pregibon, A. Rogers, F.S. Hancock, A language for extracting signatures from data streams, in *Proc. Sixth Intl. Conf. on Knowledge Discovery and Data Mining* (2000), pp. 9–17
10. DSKI. Dski—the data stream kernel interface. <http://www.ittc.ku.edu/dastream>
11. Endace. Endace web page. <http://www.endace.com>
12. N.C. Hutchinson, L.L. Peterson, Design of the x-Kernel, in *Proceedings of the SIGCOMM'88 Symposium*, Stanford, Calif. (1988), pp. 65–75
13. V. Jacobson, C. Malan, S. McCanne, Libpcap and tcpdump home page. <http://www.tcpdump.org/>
14. S. Madden, M. Franklin, Fjording the stream: an architecture for queries over streaming sensor data, in *Intl. Conf. on Data Engineering* (2002)
15. G.R. Malan, F. Jahanian, An extensible probe architecture for network protocol performance measurement, in *ACM SIGCOMM'98* (1998)
16. S. McCanne, V. Jacobson, The BSD packet filter: a new architecture for user-level packet capture, in *USENIX Winter* (1993), pp. 259–270
17. Narus. Narus platform. <http://www.narus.com/w/solutions/platform/>
18. Niksun. Product solutions. <http://www.niksun.com/product-list.html>
19. P. Seshadri, M. Livny, R. Ramakrishnan, The design and implementation of a sequence database system, in *Proc. of the 22nd VLDB Conf.* (1996)
20. J.W. Stewart III, *BGP4: Inter-Domain Routing in the Internet* (Addison-Wesley, Reading, 1999)

21. M. Sullivan, A. Heybey, Tribeca: a system for managing large databases of network traffic, in *Proc. USENIX Annual Technical Conf.* (1998)
22. D. Terry, D. Goldberg, D. Nichols, B. Oki, Continuous queries over append-only databases, in *Proc. ACM SIGMOD Conf.* (1992), pp. 321–330
23. P. Tucker, D. Maier, T. Sheard, L. Fegaras, Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.* **15**(3), 555–568 (2003)
24. S. Waldbusser, RFC 2819: remote monitoring management information base (2000)

High-Performance XML Message Brokering

Yanlei Diao and Michael J. Franklin

1 Introduction

For distributed environments including Web Services, data and application integration, and personalized content delivery, XML [5] is becoming the common *wire format* for data. In this emerging distributed infrastructure, XML *message brokers* [1, 2] will play a key role as central exchange points for messages sent between applications and/or users. The context in which such a message broker operates is shown in Fig. 1. Users (equivalently, applications, or organizations) subscribe to the message broker by providing profiles expressing their data interests. After arriving at the message broker, these profiles become “standing queries,” which are executed on all incoming data. Data sources publish their data by pushing streams of XML messages to the broker. The broker delivers to each user the messages that match his data interests; these messages are presented in the required format of the user.

There are three main functions provided by XML message brokers: *filtering*, *transformation*, and *routing*. Filtering matches XML messages to a large set of queries that represent the data interests of specific users. Transformation restructures matched messages according to user-specific requirements. Routing involves the transmission of the customized data to the relevant users.

This work has been supported in part by the National Science Foundation under the ITR grants IIS0086057 and SI0122599 and by Boeing, IBM, Intel, Microsoft, Siemens, and the UC MICRO program.

Y. Diao (✉)

Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, USA
e-mail: yanlei@cs.umass.edu

M.J. Franklin

Computer Science Division, University of California, Berkeley, Berkeley, CA, USA
e-mail: franklin@cs.berkeley.edu

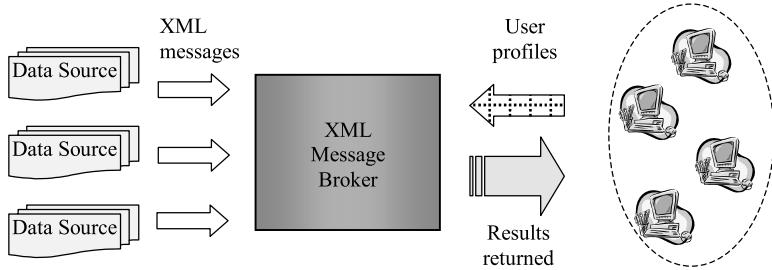


Fig. 1 Overview of XML message brokering

For XML filtering, user queries are usually expressed in a language such as XPath 1.0 [11], which is used to specify constraints over both structure and content using path expressions. An earlier project, XFilter [3], pioneered the use of event-based parsing and *Finite State Machines* (FSMs) for fast structure-oriented XML filtering. In XFilter, the structural part of each path expression is converted into an FSM by mapping location steps of the expressions to machine states. For each arriving XML document, the events raised during parsing are used to drive such FSMs through their various transitions. By creating a separate FSM for each distinct path expression, however, XFilter fails to exploit commonality among the path expressions, and thus, may perform redundant work.

Based on this insight, we have developed “YFilter”, an XML filtering system aimed at providing efficient filtering for large numbers (e.g., 10s or 100s of thousands) of path queries. The key innovation in YFilter is a *Nondeterministic Finite Automaton* (NFA)-based representation of path expressions which combines all queries into a single machine. YFilter exploits commonality among path queries by merging the common prefixes of the paths so that they are processed at most once. The NFA-based implementation also provides additional benefits including a relatively small machine size, flexibility in dealing with diverse characteristics of data and queries, incremental machine construction, and ease of maintenance.

An important challenge that arises due to the shared structure matching is the handling of value-based predicates that address contents of elements. We have developed two alternative approaches to handling such predicates. Following the heuristic of evaluating “cheap” predicates early in the execution as in relational systems, one approach evaluates predicates as soon as the addressed elements are read from a message. The other approach delays predicate evaluation until the structure of a path expression has been entirely matched.

The XML filtering solution described so far has focused on the matching of messages to large numbers of queries, but has not addressed the customization of output. Our work on XML transformation is aimed at developing the next level of functionality, i.e., transforming the XML messages on a per-query basis, in order to provide customized data delivery. User queries specifying transformation requirements can be written using a subset of XQuery [4]. To support such queries, we leverage the YFilter shared path matching engine [12], and develop alternatives for

building customization functionality on top of it. In particular, we have developed three techniques that differ in the extent to which they push work down into the path matching engine. Due to an inherent tension between shared path matching and customized result generation, aggressive path sharing requires more sophisticated post-processing. To reduce the cost of such post-processing, we have developed provably safe optimizations based on query and DTD (if available) inspection that are able to simplify the post-processing of individual queries, and a set of techniques that enable the sharing of such post-processing across multiple queries.

Roadmap

The remainder of this chapter is organized as follows. Section 2 describes the broker architecture. Sections 3 and 4 present our solutions to XML filtering and transformation. Section 5 covers the related work and Sect. 6 presents conclusions.

2 Architectural Overview

Our XML message broker architecture is shown in Fig. 2. The primary inputs are the queries that represent user profiles and the XML messages themselves.

Queries become active as soon as they arrive at the message broker. Inside the broker, an arriving query is parsed for use by the Query Processor, where the execution plan of the new query is merged with the existing queries without recompiling any of them.

Incoming messages are filtered and transformed on-the-fly for the entire set of queries. These messages need not conform to DTDs (*Document Type Definitions*) or XML schemas but such conformance can be exploited.¹ Internally, the broker runs an incoming message through an event-based XML parser. Parsing events are passed to the query processor to drive the query execution. They are also used to incrementally construct a node-labeled tree, which provides materialization of the parsed message for need of some query processing. The nodes are assigned integer identifiers according to a pre-order traversal of the tree.

The query processor produces output in an intermediate format that contains identifiers of nodes in the parsed XML message organized for efficient translation into customized output messages. The intermediate output of the query processor is fed to the “message factory”, which combines the element tags in queries with the intermediate output and forwards the resulting messages for delivery.

The query processor is the focus of our research work. The foundation of the query processor is a shared path matching engine. The shared path matching approach that this engine implements is described in the next section.

¹XML Schemas provide richer information than DTDs (e.g., robust and extensible datotyping). Since the structural information we exploit is provided by both types of definition, XML schemas and DTDs are used under the same conditions in this work.

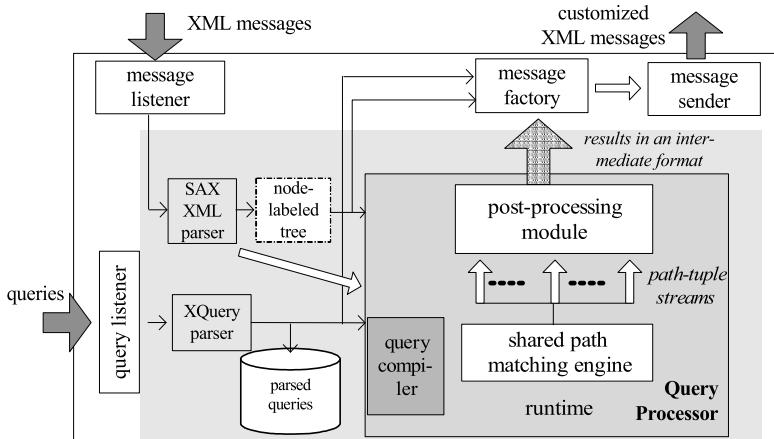


Fig. 2 XML message broker architecture

3 Shared Processing of Path Expressions

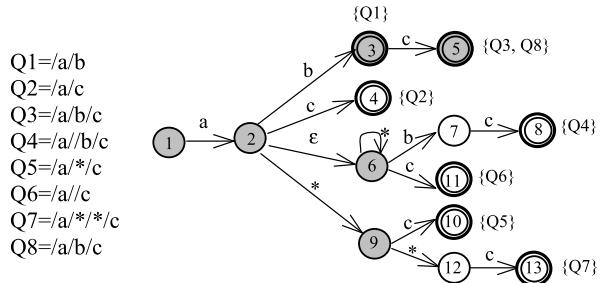
In our work, the solution to efficient matching of path expressions is based on two key observations. First, any single path expression written using the axes (“`/`”, “`//`”) and node tests (element name or “`*`”) can be transformed to a regular expression. Thus, there exists an FSM that accepts the language described by such a path expression [18]. Second, if two path expressions share a sub-expression, the language described by that sub-expression can be accepted by a single FSM. For large-scale filtering of XML data, exploiting such commonality is the key to efficiency and scalability. In this section, we describe our shared path processing approach based on a combined FSM.

3.1 An NFA-Based Model with an Output Function

In YFilter, all of the path queries written using the axes and node tests described above are combined into a single FSM that takes a form of *Nondeterministic Finite Automaton* (NFA). All common prefixes of the paths are represented only once in the NFA.

Figure 3 shows an example of such an NFA, representing eight queries (the process for constructing such a machine is described in the following section). A circle denotes a state. Two concentric circles denote an accepting state; such states are also marked with the IDs of the queries they represent. A directed edge represents a transition. The symbol on an edge represents the input that triggers the transition. The special symbol “`*`” matches any element. The symbol “`ε`” is used to mark a transition that requires no input. In the figure, shaded circles represent states shared by queries. Note that the common prefixes of all the queries are shared. Also note that

Fig. 3 Path queries and a corresponding NFA



the NFA contains multiple accepting states. While each query in the NFA has only a single accepting state, the NFA represents multiple queries. Identical (and structurally equivalent) queries share the same accepting state (recall that at the point in the discussion, we are not considering predicates).

This NFA can be formally defined as a *Moore Machine* [18]. The output function of this Moore Machine is a mapping from the set of accepting states to a partitioning of identifiers of all queries in the system, where each partition contains the identifiers of all the queries that share the accepting state.

Some Comments on Efficiency

A key benefit of using an NFA-based approach is the tremendous reduction in machine size it affords. It is reasonable to be concerned that using an NFA-based model could lead to performance problems due to (for example) the need to support multiple transitions from each state. A standard technique for avoiding such overhead is to convert the NFA into an equivalent DFA [18]. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number of states. But, as pointed out in [15], this explosion can be avoided in many cases by placing restrictions on the set of DTDs (i.e., document types) and queries supported, and lazily constructing the DFA.

The experimental results on the performance of the NFA-based approach reported in our earlier work [12], however, indicate that such concerns about NFA performance in this environment are unwarranted. In fact, in the YFilter system, path evaluation (using the NFA) is sufficiently fast, that it is not the dominant cost of filtering in many cases. Rather, other costs such as document parsing and result collection are often more expensive than the basic path matching. Thus, while it may be possible to further improve path matching speed, we believe that the substantial benefits of flexibility and ease of maintenance provided by the NFA model outweigh any marginal performance improvements that remain to be gained by even faster path matching.

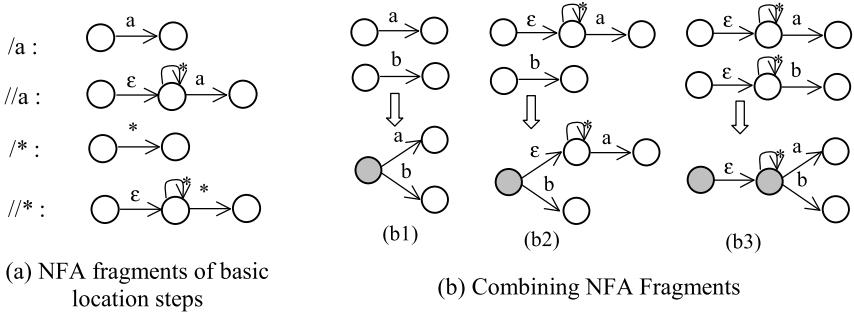


Fig. 4 NFA fragments for location steps, and examples of merging NFA fragments

3.2 Constructing a Combined NFA

Having presented the basic NFA model used by YFilter, we now describe an incremental process for NFA construction and maintenance. The shared NFA shown in Fig. 3 was the result of applying this process to the eight queries shown in that figure.

Figure 4(a) shows four basic location steps in our subset of XPath, and the directed graphs, called *NFA fragments*, that correspond to these location steps. The NFA for a path expression, denoted as NFA_p , can be built by concatenating all the NFA fragments for its location steps. The final state of this NFA_p is the (only) accepting state for the expression.

NFA_p s are combined into a single NFA as follows: There is a single initial state shared by all NFA_p s. To insert a new NFA_p , we traverse the combined NFA until either (i) the accepting state of the NFA_p is reached, or (ii) a state is reached for which there is no transition that matches the corresponding transition of the NFA_p . In the first case, we make that final state an accepting state (if it is not already one) and add the query ID to the query set associated with the accepting state. In the second case, we create a new branch from the last state reached in the combined NFA. This branch consists of the mismatched transition and the remainder of the NFA_p . Figure 4(b) provides three examples of this process. Note from (b2) that in the NFA fragment for a location step with “//”, the ϵ -transition is needed so that when combining NFA fragments representing “//” and “/” steps, the resulting NFA accurately maintains the different semantics of both steps.

It is important to note that because NFA construction in YFilter is an incremental process, new queries can easily be added to an existing system. This ease of maintenance is a key benefit of the NFA-based approach.

3.3 Executing the NFA

Following the XFilter approach, the NFA is executed in an event-driven fashion. When an XML document arrives to be filtered, it is parsed with an event-based

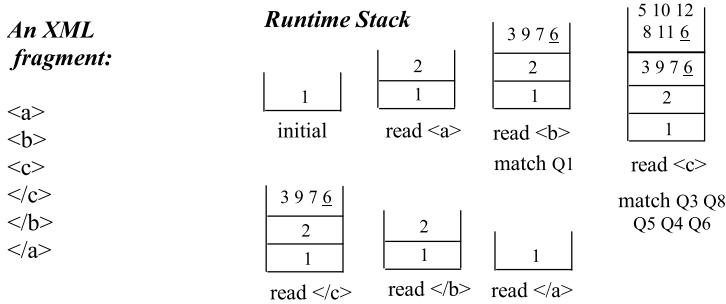


Fig. 5 An example of the NFA execution

parser; each time a new element or the end of an element is encountered, an event is raised. The start-of-element events trigger various transitions in the NFA. Since the machine is non-deterministic, many states can be active simultaneously. In addition, the nesting of XML elements requires that when an “end-of-element” event is raised, NFA execution must backtrack to the states it was in when the corresponding “start-of-element” was raised. A stack mechanism is used to trace multiple active states and enable backtracking.

Figure 5 shows the evolution of the contents of the stack as an example XML document is parsed. Each state in the stack is represented by its state ID, as shown in Fig. 3. On receiving a start-of-element event, the execution engine follows all matching transitions from all currently active states. For each active state, four checks are performed. First, if a transition marked by the incoming element name is present, the next state is added to the set of new active states. A transition marked by the “*” symbol is checked in the same way. Then, the state itself is added to the set if it has a self-loop, which is represented by an underlined state ID in Fig. 5. Finally, if an ϵ -transition is present, the state after the ϵ -transition is processed immediately according to these same rules. The interested reader is referred to [12] for more details on the NFA-based processing of path expressions.

Finally, it is important to note that, unlike a traditional NFA, whose goal is to find one accepting state for an input, the NFA execution in this work must continue until all potential accepting states have been reached. This is because all queries that match the input document need to be found for the purpose of XML filtering.

3.4 Predicate Evaluation

The discussion so far has focused on the structure matching aspects of YFilter. In an XPath expression, however, predicates can be applied to address properties of elements, such as the text data and attributes. In this section, we focus on these value-based predicates and briefly describe the techniques used in YFilter to evaluate them. The discussion on the evaluation of nested path expressions is postponed until Sect. 4.

Given the NFA-based model for path-matching, an intuitive approach to supporting value-based predicates would be to extend the NFA by including additional transitions to states that represent the successful evaluation of the predicates. Unfortunately, such an approach could result in an explosion of the number of states in the NFA, and would destroy the sharing of path expressions, the key advantage of using an NFA.

YFilter uses a separate selection operator that evaluates value-based predicates by interacting with the NFA-based processing of path expressions. Traditional relational query processing uses the heuristic of pushing selections down in the query plan to avoid unnecessary future work. Following this intuition, we developed an approach called *Inline* that processes value-based predicates as soon as the elements that those predicates address are matched during structure matching. To do so, predicates applied to a location step are associated with the state representing this location step in the NFA; these predicates are evaluated using selection immediately when this state is visited in the NFA execution. For each query, bookkeeping information is maintained, indicating which predicates of that query can be satisfied by a particular element. When an accepting state is reached, the bookkeeping information for the queries of that state is checked, and those queries for which all the predicates can be successfully evaluated are returned as matches.

We also developed an alternative approach, called *Selection Postponed* (SP) that waits until an entire path expression is matched during structural matching, and at that point applies all the value-based predicates for the matched path. In SP, the predicates are stored with each query. When an accepting state is reached in the NFA, selections are performed in bulk for each query associated with the state. If all predicates of a query evaluate to true, then the query is satisfied. A more detailed description of the above two approaches is provided in [12].

3.5 Performance Results for Shared Path Processing

We have performed a detailed performance study of our YFilter implementation. In the study, we compared the performance of the NFA-based path matching approach in YFilter, the FSM-per-query approach used by XFilter, and a hybrid approach that exploits a reduced degree of shared path processing. We also investigated the tradeoffs between the *Inline* and SP approaches to value-based predicates. The full results are reported in an earlier paper [12]. Here, we summarize those results as follows:

- YFilter can provide order of magnitude performance improvements over both XFilter and the hybrid approach. In fact, as discussed earlier, path processing using YFilter is sufficiently fast that in many cases it is outweighed by other costs for XML filtering such as document parsing and result collection.
- The NFA-based approach is robust and efficient under query workloads with varying proportions of “//” operators and “*” operators. This is important because it is these operators that introduce non-determinism into the path matching process.

- The NFA-based approach was also shown to perform well using a number of DTDs with different characteristics.
- The maintenance cost (i.e., as queries are added and removed) of the NFA structure is small, due to the incremental construction that a nondeterministic version of an FSM enables and due to the sharing of structure inherent in the NFA approach.
 - For value-based predicates, the SP approach was found to perform much better than the Inline approach. The Inline approach suffers from high execution complexity (e.g., the bookkeeping overhead). Moreover, early predicate evaluation is not effective in eliminating future work of structure matching or predicate evaluation, due to the shared nature of path matching and the effect of recursive elements in the presence of “//” operators in path queries. In contrast, SP is relatively simple and uses path matching to prune the set of queries for which predicate evaluation needs to be considered, thus achieving a significant performance gain.

4 Customized Result Generation

The XML filtering solution as presented so far has focused on the matching of large numbers of queries, but has not addressed the customization of output. Support for such customization can significantly increase the complexity of processing. This section presents approaches that leverage the YFilter path matching engine for result customization; in particular, they use the path engine for shared path matching and perform post-processing on the path engine output to extract the specific XML elements needed for the customized results.

4.1 Input and Output Specifications

We focus on user queries written in a subset of XQuery. Consider **Query 1** below, which is based on the Book DTD from the XQuery use cases [7].

```

<sections>
{
  for $s in $msg//section
  where $s//figure/title = "XML processing"
  return <section>
    { $s/title }
    { $s//figure }
  </section>
}
</sections>
```

This query specifies that for each section containing a figure whose title is “XML processing”, a “section” element containing the title of that section and all of its

figures should be returned. Note that in a document conforming to the Book DTD, section elements may contain other sections as well as figures and other elements. This query requires results to be returned for all distinct sections matching the query in the same order that the matching sections appear in the message.

More specifically, the queries we consider consist of a single *FLWR* (i.e., *For-Let-Where-Return*) expression, optionally enclosed in elements defined by *constant tags*. The FLWR expression contains the following clauses in the specified order:

- A `for` clause containing a *variable name* and a *path expression*;
- An optional `where` clause that contains a set of conjunctive predicates, each of which takes a form of a triplet: *path expression, op, constant*;
- A `return` clause that contains interleaved *constant tags* and *path expressions*, where all constant tags have a matching close tag.

The current implementation in this work does not support the `let` clause.

The semantics of such queries is as follows: The `for` clause creates an *ordered sequence* of variable bindings to *distinct* element nodes. The `where` clause, if present, restricts the set of bindings passed to the `return` clause. The `return` clause is invoked once for each variable binding. At each invocation of the `return` clause, tags cause the construction of new XML fragments and path expressions select nodes from the current variable binding. The final result of the FLWR expression is an *ordered sequence* of the results of these invocations.

For conciseness, the path expression in a `for` clause is referred to as the *binding path*, those in a `where` clause as *predicate paths*, and those inside a `return` clause as *return paths*. In this work, we consider predicate and return paths that are relative to the binding path of that query (i.e., they are prefixed by the variable name used in the binding path). As in our previous work on XML filtering, a path expression consists of a sequence of location steps; location steps can contain the child “`/`” or the descendent “`//`” axis, and element name tests. Path expressions containing such location steps are referred to as *navigation paths*. We also allow location steps to contain predicates that compare the attributes or text data of elements to a constant, referred to as *value-based predicates*. In this work, binding paths can contain an arbitrary number of value-based predicates in any location step. A predicate path is a navigation path with a value-based predicate attached to the last location step (which is effectively a *nested path expression* imposed on its binding path). A return path is simply a navigation path.²

As stated in Sect. 2, the output of the query processor is an intermediate representation that is passed on to the message factory component of the broker. In this representation, the nodes selected from the message are organized into a sequence of groups, such that each group corresponds to a single invocation of the `return` clause. Inside a group, nodes are contained in a sequence of lists. The sequencing of lists corresponds to the ordering of the return paths in the `return` clause. Each list

²The approaches we describe in this section can be extended to support more general XQuery scenarios. This extension is our future work, and will not be discussed further here.

contains the nodes matching the return path in their document order. For example, the output of Query 1 would have the following format:

```
.....  
sectioni : [titlei1], [figurei1, ...]  
sectioni+1 : [title(i+1)1], [figure(i+1)1, ...]  
.....
```

where section_i represents a group, and the numbering $\dots, i, i + 1, \dots$ represents the ordering of those groups. The sequence inside a group consists of a list of identifiers of title nodes (in the above example there is only a single title per section) followed by a list of identifiers of figure nodes. This representation is referred to as the *groupSequence-listSequence* format.

4.2 Query Processor Details

In the YFilter system, as shown in Fig. 2, the query processor consists of two main runtime components: a shared path matching engine and a post-processing module. Given a parsed query, the compiler in the query processor inserts navigation paths from the query into the shared path matching engine, and adds the execution plan for the remainder of the query to the post-processing module. For an incoming message, the shared path matching engine takes the parsing events to match its contained navigation paths. The post-processing module further processes the output of the path matching engine to generate customized results in the *groupSequence-listSequence* format.

Our solutions to customized result generation are developed in the context of the particular output format provided by the path matching engine. For a navigation path matched by an incoming message, the path engine delivers a stream of “path-tuples” each of which represents a unique match of this path. A path-tuple contains one field per location step in the path, and the value of the field is the identifier of the element node bound to the location step. When multiple paths are matched by a message, the path engine delivers its output as streams of path-tuples, one stream for each path.

Figure 6(a) shows a node-labeled tree for a message fragment, where nodes are annotated with their assigned ids. Path-tuple streams that are output from the path engine for different paths are illustrated in Fig. 6(b). Take the stream for the path “//section//figure”. It contains three path-tuples. Each path-tuple contains two node ids, representing a unique combination of the two location step bindings.

The path engine guarantees that path-tuples in each stream are produced such that the node ids in the last field of the path-tuples appear in monotonically increasing order. This stream order is exploited in the processing algorithms as described in the following sections. It is worth noting that ordering on other fields of path-tuples is not guaranteed, however.

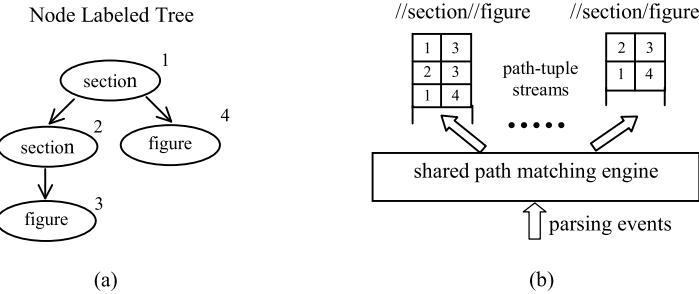


Fig. 6 An example of the NFA execution

The following three subsections present three different query processing approaches that differ in the extent to which they exploit the path matching engine. In all of them, post-processing of the matching engine output is done via query plans using relational-style operators. Such a post-processing plan is created per XQuery query (i.e., the post-processing phase is not shared). The discussion on shared post-processing is postponed to Sect. 4.7.

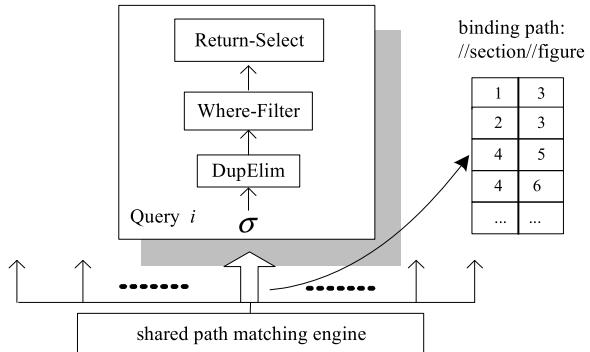
Much of the subtlety of developing solutions to this problem arises from the inherent tension between shared processing at the lower level (which is essential for good performance) and customized query result generation. The matching engine returns the path-tuples in a stream in a single, fixed order to all queries that include the corresponding path. The paths, however, may be used quite differently by the various queries, and thus potential inconsistencies such as unintended duplicates or ordering problems can arise with aggressive path sharing (these cases will be discussed in detail shortly). In the following, we describe these approaches in order of increasing path sharing, and focus on how the additional complications raised by increased sharing are addressed.

4.3 Shared Matching of “for” Clauses

The first approach uses the path matching engine to process only binding paths (i.e., paths that appear in `for` clauses). In this approach, the navigation part of the binding path is inserted from each query into the engine. Then, during the processing of a message, the output of the engine for each path is directed to the post-processing plans for its corresponding queries. This approach is referred to as *PathSharing-F*. Consider **Query 2** below:

```
<figures>
{
    for      $f in $msg//section[@id ≤ 2]//figure
    where   $f/title = "XML processing"
    return <figure>
```

Fig. 7 A query plan using PathSharing-F



```

    { $f/image }
  
```

/figure)

}

/figures)

Figure 7 highlights the post-processing plan for this query under *PathSharing-F*. In the figure, the multiple arrows above the matching engine represent the streams of path-tuples (note that queries that have a common binding path share a common stream). The thick arrow denotes the stream used by Query 2, which contains the path-tuples matching the binding path “`//section//figure`”. In the following, the last field of these path-tuples is referred to as the *binding field* because they contain the ids of the nodes that are actually bound by the binding paths. These nodes are referred to as the *BoundNodes*. The box above the thick arrow contains the post-processing execution plan. The operators in this plan are, from bottom-up:

Selection

A selection operator is placed at the bottom of a query plan to evaluate any value-based predicates (e.g., comparisons of the attributes or text data of elements to a constant) attached to a binding path. The evaluation is done for each path-tuple by checking predicates against the nodes referenced by the path-tuple. Selection emits only those path-tuples for which all predicates evaluate to True.

Duplicate Elimination (DupElim)

The XQuery specification requires that duplicate nodes bound to a path be eliminated based on the node identity [4]. Accordingly, we define duplicates in the stream for a binding path as path-tuples that contain the same node id in the binding field. Such duplicates arise when multiple path-tuples in a stream reference the same

BoundNode. For example, consider Query 2 and the XML fragment:

```
<section id = “1”><section id = “2”><figure><title>
XML processing</title></figure></section></section>
```

The matching engine outputs two path-tuples for the binding path. The first corresponds to “`<section id = “1”> <figure>`” and the second to “`<section id = “2”> <figure>`”. These two path-tuples reference the same BoundNode, so the second could cause redundant work and produce a duplicate result.

The DupElim operator avoids these problems by ensuring that each BoundNode is emitted at most once. In this case, a simple scan-based DupElim operator can be used because as described in the previous section, path-tuples in the stream are ordered by their binding field. It should be noted, however, that DupElim cannot be pushed before the selection because it is not known which (if any) of the path-tuples referencing the same BoundNode will pass the selection.

Where-Filter

This operator evaluates the `where` clause on each path-tuple until a predicate in the `where` clause evaluates to False or the entire `where` clause evaluates to True. Path-tuples in the latter case are emitted. For each path-tuple, a predicate path is evaluated with a tree search routine that uses a depth-first search in the sub-tree of the parsed message rooted at the BoundNode of the path-tuple. The search routine for a path returns True as soon as any node satisfying the predicate is found.

Return-Select

This operator applies the `return` clause to the BoundNodes of the path-tuples that survive the Where-Filter. It uses the tree search routine to select nodes for each return path. Unlike the Where-Filter, however, the tree search routine here must retrieve all nodes matching a return path rather than stopping at the first match. In addition, it generates results in the *groupSequence-listSequence* format. Each input path-tuple causes the creation of a new group. The ordering of return paths in the query defines the sequence of lists within each group. For each list, the nodes selected for the corresponding return path are placed in the order that they appear in the message.

Recall that the results of a FLWR expression must be ordered in accordance with the order of the variable bindings of the `for` clause. Since the stream for the binding path is ordered in this way, and the remaining processing steps do not change that order, we are assured that the order produced by *PathSharing-F* is correct.

4.4 Shared Matching of “Where” Clauses

PathSharing-F only uses the path matching engine to process binding paths. The next approach, *PathSharing-FW*, in addition pushes the navigation part of predicate paths from the where clause into the matching engine to exploit further sharing. Recall that predicate paths are defined to be relative to the binding paths. Since the matching engine treats all paths as being independent, the predicate paths must be first extended by pre-pending their corresponding binding path. For example, consider **Query 3** below:

```

<sections>
{
  for $s in $msg//section
  where $s/title = "XML"
  and   $s/figure/title = "XML processing"
  return <section>
        { $s//section/title }
        { $s/figure }
      </section>
}
</sections>
```

The first predicate path “/title” is transformed into “//section/title” and the second becomes “//section/figure /title”. These extended predicate paths, along with the binding path, are inserted into the matching engine. Note that since common prefixes of paths are shared in the matching engine, the extension of these paths does not add significantly to their processing cost.

As in *PathSharing-F*, the path-tuple streams for each query are then post-processed by a query plan that executes the remaining portion of that query. This arrangement is shown in Fig. 8. The stream corresponding to a binding path is passed through a selection operator and a DupElim operator as before. The output of the DupElim operator is then matched with the streams corresponding to the predicate paths. The path-tuples resulting from the matching process are piped to a Return-Select that works as described before.

In *PathSharing-FW*, the Where-Filter of *PathSharing-F* is replaced by a left-deep tree of *semijoins* with the binding path stream as the leftmost input. Recall that the predicate paths are extended by pre-pending them with the corresponding binding path. Thus, the common field on which each semijoin will match is the *binding field*, i.e., the last common field between the binding path tuples and the predicate path tuples. The result of a semijoin, therefore, is a stream containing only those binding path tuples that have matching predicate path tuples. Figure 8 shows an example for the leftmost semijoin.

The semijoin operators can be implemented using a simple merge-based algorithm, if it is known that the predicate path streams are delivered in monotonically increasing order of BoundNode id. In general, however, there are cases where such

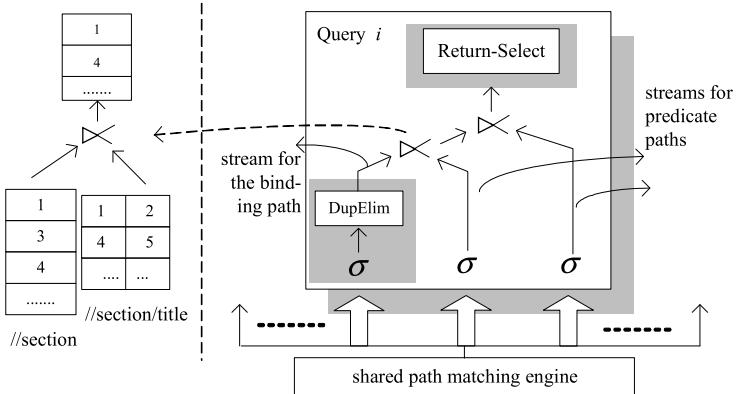


Fig. 8 A query plan using PathSharing-FW

ordering cannot be assumed. Consider the execution of Query 3, when applied to the following XML fragment:

```
<section><section><figure><title>XML processing
</title></figure></section><figure><title>
XML processing</title></figure></section>
```

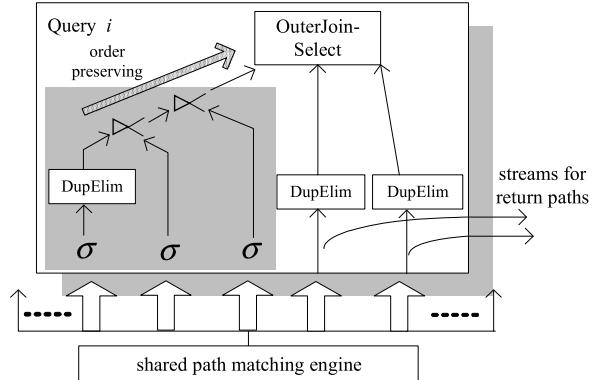
In this case, the stream for the predicate path “//section/figure/title” would contain a path-tuple corresponding to “section₂ figure₁ title₁” followed by a path-tuple corresponding to “section₁ figure₂ title₂”, where the subscript indicates the first or the second occurrence of the element name. This stream is not properly ordered by the binding field (i.e., “section”). In such cases, since the binding path stream is ordered properly, we can use a hash-based implementation of semijoin where the binding path stream is used as the probing stream. Sufficient conditions for determining when the more efficient merge-based approach can be used are discussed in Sect. 4.6. Note, however, that both approaches order the output correctly, resulting in semantics identical to those provided by *PathSharing-F*.

4.5 Shared Matching of “Return” Clauses

The third alternative approach, *PathSharing-FWR*, aims at further increasing sharing by also pushing the return paths into the path matching engine. Return paths differ from predicate paths in that they do not constrain the set of matching binding path tuples so the semijoin approach cannot be used for them. Instead, *outer-join* semantics are required.

This particular work requires a slightly more specialized operator than a generic outer-join, however, because results must be generated in the *groupSequence*-

Fig. 9 A query plan using PathSharing-FWR



listSequence format. Thus, we have implemented our own n-way outer-join operator, called *OuterJoin-Select*. As Fig. 9 shows, OuterJoin-Select takes as its leftmost input, the binding path stream resulting from the semijoins of the *PathSharing-FW* approach. It performs left outer joins on the binding field with each of the return path streams. Generation of the results in the required format is performed as part of the outer join processing. Each path-tuple in the binding path stream causes the creation of a new group. The outer join between this path-tuple and a return path stream results in a new list within the group, containing the node ids in the last field in the return path tuples that have matched the binding path tuple. If no such matches are found, an empty list is kept in the group for this return path. The issue of hash- versus merge-based implementation of OuterJoin-Select is similar to that of semijoins.

Note from Fig. 9 that DupElim operators are required on each of the return path streams to prevent duplicate results from being generated by OuterJoin-Select. Here, the notion of duplicates is defined on the combination of the *binding field* and the last field of the path-tuple, called the *return field*. Recall that a return path stream is always ordered by the return field. If it also arrives ordered by the binding field, a scan-based approach suffices for DupElim. Otherwise, hashing is used.

As can be seen, *PathSharing-FWR*, the approach that exploits path sharing to the fullest extent, requires the most sophisticated post-processing. As mentioned earlier, this complexity results from the tension between shared path matching and result customization. It is important to note that this problem cannot be easily solved in the path matching engine. Consider a path expression that is the binding path in one query and a return path in another. In this case, the path-tuple stream produced for that path expression will be used (by different queries) as two different types of streams. Since the two types of streams have different notions of duplicates, duplicate elimination cannot be done in the engine, but must be done in a usage-specific manner during post-processing. Similar issues arise with the ordering of path-tuples expected by the different uses of the stream.

4.6 Simplifying Post-Processing

Duplicates and stream ordering are two fundamental issues that complicate post-processing for customized result generation. With additional knowledge, however, it is sometimes possible to infer cases when duplicates cannot arise, or when path-tuples will arrive in a needed order. In the first case, DupElim operators can be removed from the post-processing plans. In the second case, cheaper scan or merge-based operator implementations can be used in place of the more expensive hash-based ones.

We have derived a set of sufficient conditions that enable the detection of some situations where post-processing can be simplified. These conditions involve the presence of “//” axes in queries, and the potential for recursive elements (i.e., elements that have the same element name and contain each other) in the messages. The first type requires examining the queries, and the second can be checked by examining a DTD, if present. The set of sufficient conditions and a detailed description on how they are applied to optimize post-processing plans are provided in [13].

4.7 Shared Post-Processing

A common feature of the three approaches to sharing path matching among queries is that they all require a separate post-processing plan for each query. We have also developed an initial set of techniques that can further improve sharing by allowing some of the post-processing work to be shared across related but non-identical queries, in particular, ones that have path expressions (and hence, path-tuple streams) in common. Similar in spirit to some techniques proposed for shared *Continuous Query* (CQ) processing over (typically non-XML) data streams [9, 10, 17, 20, 22], the techniques developed in this work are highly tailored for the unique set of operators used and a specific data flow through them for processing large numbers of XQuery queries. Examples of these techniques include query rewriting, shared GroupBy for OuterJoin-Select, selection-DupElim pull up, and shared selection. The interested reader is referred to [13] for details of these techniques.

4.8 Performance Results for Customized Result Generation

The experiments reported in [13] have compared the performance of the three alternatives for exploiting YFilter’s shared path matching engine with and without optimizations. The experiments also investigated the performance of a suite of techniques to share the post-processing among queries. These results can be summarized as follows:

- *PathSharing-FWR* when combined with optimizations based on queries and DTD usually provides the best performance. This approach is the most aggressive of the three in terms of path sharing. Without optimizations, however, *PathSharing-FWR* performs quite poorly, due to high post-processing costs.
- Optimization of query plans using query information improves the performance of all alternatives, and the addition of DTD-based optimizations improves them further.
- For non-recursive data, DTD-based optimizations can remove all DupElim operators and turn all hash-based implementations to merge/scan-based. Recursive data, however, stresses the post-processing of queries containing “//” axes and limits the effectiveness of optimizations.
- Finally, experiments on extending *PathSharing-FWR* with shared post-processing showed excellent scalability improvements, allowing the processing of 100,000 queries in less than half a second.

5 Related Work

Our work on XML message brokering is related to Continuous Query (CQ) processing, publish/subscribe, XML filtering, XML stream processing, and multi-query processing.

CQ systems support shared processing of multiple standing queries over (typically non-XML) data streams. The concept of expression signatures was introduced by *TriggerMan* [17]. Using such expression signatures, *NiagaraCQ* [9, 10] incrementally groups query plans, and *CACQ* [22] supports the sharing of physical operators among tuples. *OpenCQ* [20] uses grouped triggers for CQ condition checking. Our techniques for sharing post-processing, though similar in spirit to those used in some of these systems, are developed particularly for XQuery processing.

Publish/subscribe systems, e.g., *Le Subscribe* [14] and *Xlyeme* [23], match incoming events with a very large number of subscriptions each of which is typically a set of conjunctive predicates. These systems use restricted query languages and data structures tailored to the query languages to achieve high system throughput.

A number of XML filtering systems have been developed to efficiently match a large set of path queries with streaming documents. *XFilter* [3] builds a Finite State Machine (FSM) for each path query and employs a query index on all the FSMs to process all queries simultaneously. *XTrie* [8] supports shared processing of the common substrings of path expressions which only contain parent-child operators. In [15], all path expressions are combined into a single DFA, resulting in good performance but with significant limitations on the flexibility of the approach. *YFilter* and *Index-Filter* are compared through a detailed performance study in [6]. *Match-Maker* [19] supports shared tree pattern matching using disk-resident indexes on the tree patterns, with limited filtering performance. *XPush* [16] builds a pushdown automaton for a subset of tree-pattern queries, sharing both path navigation and predicate evaluation among them. It requires some precomputation of the machine

to achieve good performance. As stated previously, these systems only provide the lowest level of functionality required by XML message brokers.

In the context of XML stream processing, some other recent work uses transducer based mechanisms for processing path expressions with qualifiers [21] or XQuery containing FLWR expressions [24]. These approaches, however, are developed for single query processing.

Multi-query processing [25–27] considers small numbers of queries (e.g., 10s) and uses heuristics to approximate the optimal global plan. In contrast, high-volume XML message brokering needs to handle sets of queries orders of magnitude larger in a dynamic environment. Thus, scalability of the approach and incremental construction of query plans are the major concerns unique to our work.

6 Conclusions

In this chapter, we presented our approaches to shared processing of queries for XML filtering and transformation in the context of high-performance XML message brokering. For XML filtering, we developed an NFA-based shared path matching engine that provides flexibility and excellent performance by exploiting overlap of path expressions. To support the customization of output, we developed three different ways of exploiting the shared path matching engine. The most aggressive of the three in terms of path sharing is shown to perform best, when combined with optimizations based on the queries and DTD. Moreover, when post-processing of the path matching output is also shared among queries, excellent scalability can be achieved.

The results presented in this chapter, as well as other efforts cited in the related work, have demonstrated that XML message brokering is a rich source of research issues. Furthermore, as XML continues to gain acceptance in technologies such as Web Services, Event-based Processing, Application Integration, and Mobile Operators, this work will be of increasing commercial importance. As such, there are many important problems to be addressed in future work. These include incorporation of additional features such as ordering in result customization, support of data aggregation within each message and across multiple messages, and ultimately, the extension of XML message brokering in a wide-area distributed environment through the investigation of content-based routing. Research on all of these issues is currently underway.

References

1. Microsoft biztalk server (2002). <http://www.microsoft.com/biztalk>
2. Sybase financial fusion message broker (2003). <http://www.sybase.com/products/middleware/messagebroker>
3. M. Altinel, M.J. Franklin, Efficient filtering of XML documents for selective dissemination of information, in *Proc. of Int'l Conf. on Very Large Databases* (2000)

4. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, XQuery 1.0: an XML query language (2002). W3C working draft. <http://www.w3.org/TR/xquery>
5. T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, Extensible markup language XML 1.0. W3C recommendation (2000). <http://www.w3.org/TR/2004/REC-xml-20001006>
6. N. Bruno, L. Gravano, N. Koudas, D. Srivastava, Navigation- vs. index-based XML multi-query processing, in *Proc. of IEEE Int'l Conf. on Data Engineering* (2003)
7. D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, J. Robie, XML query use cases. W3C working draft (2002). <http://www.w3.org/TR/xmlquery-use-cases/>
8. C.Y. Chan, P. Felber, M.N. Garofalakis, R. Rastogi, Efficient filtering of XML documents with XPath expressions, in *Proc. of IEEE Int'l Conf. on Data Engineering* (2002)
9. J. Chen, D.J. DeWitt, J.F. Naughton, Design and evaluation of alternative selection placement strategies in optimizing continuous queries, in *Proc. of IEEE Int'l Conf. on Data Engineering* (2002)
10. J. Chen, D.J. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for Internet databases, in *Proc. of ACM SIGMOD Conf. on Management of Data* (2000)
11. J. Clark, S. DeRose, XML path language XPath—version 1.0 (1999). <http://www.w3.org/TR/xpath>
12. Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, P. Fischer, Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.* (2003)
13. Y. Diao, M.J. Franklin, Query processing for high-volume XML message brokering, in *Proc. of Int'l Conf. on Very Large Databases* (2003)
14. F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, D. Shasha, Filtering algorithms and implementation for very fast publish/subscribe, in *Proc. of ACM SIGMOD Conf. on Management of Data* (2001)
15. T.J. Green, G. Miklau, M. Onizuka, D. Suciu, Processing XML streams with deterministic automata, in *Proc. of IEEE Int'l Conf. on Database Theory* (2003)
16. A. Gupta, D. Suciu, Streaming processing of XPath queries with predicates, in *Proc. of ACM SIGMOD Conf. on Management of Data* (2003)
17. E.N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, A. Vernon, Scalable trigger processing, in *Proc. of IEEE Int'l Conf. on Data Engineering* (1999)
18. J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, 1979)
19. L.V. Lakshmanan, S. Parthasarathy, On efficient matching of streaming XML documents and queries, in *Proc. of Int'l Conf. on Extending Database Technology* (2002)
20. L. Liu, C. Pu, W. Tang, Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng.* (1999)
21. B. Ludäscher, P. Mukhopadhyay, Y. Papakonstantinou, A transducer-based xml query processing, in *Proc. of Int'l Conf. on Very Large Databases* (2002)
22. S.R. Madden, M.A. Shah, J.M. Hellerstein, V. Raman, Continuously adaptive continuous queries over streams, in *Proc. of ACM SIGMOD Conf. on Management of Data* (2002)
23. B. Nguyen, S. Abiteboul, G. Cobena, M. Preda, Monitoring XML data on the web, in *Proc. of ACM SIGMOD Conf. on Management of Data* (2001)
24. D. Olteanu, T. Kiesling, F. Bry, An evaluation of regular path expressions with qualifiers against XML streams, in *Proc. of IEEE Int'l Conf. on Data Engineering* (2003)
25. A. Rosenthal, U.S. Chakravarthy, Anatomy of a modular multiple query optimizer, in *Proc. of Int'l Conf. on Very Large Databases* (1988)
26. P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, Efficient and extensible algorithms for multi-query optimization, in *Proc. of ACM SIGMOD Conf. on Management of Data* (2000)
27. T.K. Sellis, Multiple-query optimization. *ACM Trans. Database Syst.* (1988)

Fast Methods for Statistical Arbitrage

Eleftherios Soulas and Dennis Shasha

1 Motivation

In many time series applications, the data set under study is a vast stream of continuously updated data. Often those data arrive in bursts. Many examples are found in finance, telecommunications, physics, biology, astronomy, among others [3, 18–20].

The main motivation of our project is to infer linear relationships among multiple financial variables, yielding expressions of the form $4 * \text{DJI} - 35 * \text{SNP500} = 2.4 * \text{NASDAQ}$. We use such relationships to back-test a simple trading strategy. For the most part we based our analyses on FOREX data but we also experimented with non-financial datasets. Finally, we provide software capable of being used in a variety of generalized machine learning and streaming applications in many environments. The framework we developed is named StatLearn and is based on two fundamental blocks: SketchStream to filter for relevant time series (a reimplementation of StatStream [2, 4, 16]) and LearnStream to make predictions. Thus, we provide machine learning algorithms in an on-line context.

2 SketchStream

The first step of our learning framework is to select the best candidate data from a large number of possible candidates and train on them in order to optimize the

To the memory of Lefteris's father, Christos Soulas, who supported and guided Lefteris in every step of the way.

E. Soulas · D. Shasha (✉)

New York University, 251 Mercer St, New York, NY 10012, USA

e-mail: shasha@courant.nyu.edu

E. Soulas

e-mail: es3431@nyu.edu

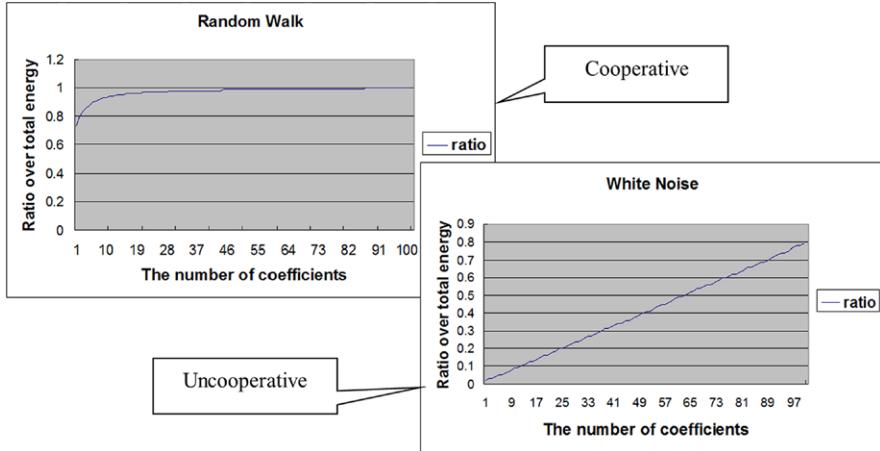


Fig. 1 Cooperative vs uncooperative data

accuracy of our results. To implement this, we find the most correlated (or anti-correlated) pairs with our target variable in an effort to try and combine all of the most statistically significant data sources that can explain the variance of the time series we are trying to predict.

Therefore, we are interested in calculating all the pairwise correlations of N_s streams in a sliding window of size sw . Doing this naively requires $O(sw * N_s^2)$ time. For applications with a many streams, this would take too long. Thus, our task is to study efficient algorithms to speed this up. We categorize the time series as cooperative or uncooperative.

- **Cooperative**—Time series that exhibit a substantial degree of regularity.

There are several data reduction techniques that can be used in order to compress long time series into a few Fourier coefficients as Fourier Transform methods [5–7], Wavelet Transforms [8, 9], Singular Value Decompositions [10], Piecewise Constant Approximations [11].

- **Uncooperative**—Time series with no such periodical regularities. An example can be the returns of a stock prices because the increase or decrease of returns do not necessarily follow any particular patterns.

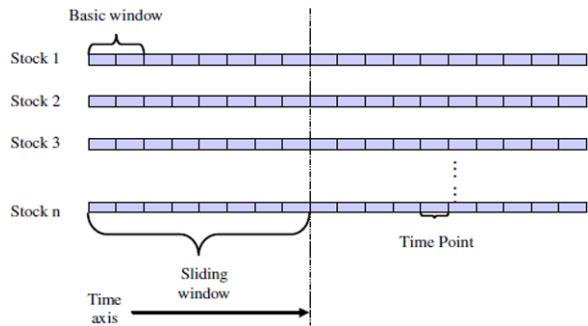
For uncooperative time series, the number of Fourier coefficients required to describe a time series grows to a large number as shown in Fig. 1.

Stock market returns ($= \frac{\text{today's_price} - \text{yesterday's_price}}{\text{yesterday's_price}}$) are “white noise-like”, that is, there is almost no correlation from one time point to the next.

For collections of time series that do not concentrate power in the first few Fourier/Wavelet coefficients, which we have termed uncooperative, we use a sketch based algorithmic framework called SketchStream.

SketchStream framework partitions time as following:

Fig. 2 Time partitioning in SketchStream



- **timepoint**—The smallest unit of time over which the system collects data, for example a second.
- **basic window**—A consecutive subsequence of timepoints over which the system maintains a compressed representation, for example two minutes.
- **sliding window**—A user-defined consecutive subsequence of basic windows over which the user wants statistics, for example, an hour. This way the user can submit queries for highly correlated stream pairs which were above a certain threshold for the past sliding window, for example between 12 noon and 1 PM, between 12:02 and 1:02, between 12:04 and 1:04, etc.

Figure 2 depicts a graphical representation of these time intervals in SketchStream.

The choice of sw and bw depends on the application under consideration because bw is the delay before results are reported. Smaller values of bw require more work.

2.1 Intuition & Guarantees of the Sketch Approach

As pointed out above, the Fourier Transform behaves very poorly for “white noise” style data. For such data, sketches are invoked.

The sketch approach, as developed by Kushikvitz et al. [14], Indyk et al. [15], and Achlioptas [12], provides a very elegant bound approach: with high probability a random mapping taking points in \mathbb{R}^m to points in $(\mathbb{R}^d)^{2b+1}$ (the $(2b+1)$ -fold cross-product of \mathbb{R}^d with itself) approximately preserves distances (with higher fidelity the larger b is). All these bounds follow from the seminal work of Johnson and Lindenstrauss [13].

The JL lemma shows that the approximation of the original distance may achieve sufficient accuracy with high probability as long as the synopsis size is larger than a given bound. This synopsis is used to filter the non-correlated time series pairs. This approach will be further addressed in the following sections.

We use correlation and distance more or less interchangeably because one can be computed from the other once the data has been normalized. Specifically, Pearson

correlation is related to Euclidean distance as follows:

$$D^2(\hat{x}, \hat{y}) = 2 * (1 - \text{corr}(x - y))$$

Given a point $x \in \mathbb{R}^m$, we compute its dot product with d random vectors $r_i \in \{1, -1\}^m$. The first random projection of x is given by $y_1 = (x * r_1, x * r_2, \dots, x * r_d)$.

We compute $2b$ more such random projections y_1, \dots, y_{2b+1} . If w is another point in \mathbb{R}^m and z_1, \dots, z_{2b+1} are its projections using dot products with the same random vectors then the median of $\|y_1 - z_1\|, \|y_2 - z_2\|, \dots, \|y_{2b+1} - z_{2b+1}\|$ is a good estimate of $\|y - z\|$. It lies within a $\theta(1/d)$ factor of $\|y - z\|$ with probability $1 - (1/2)^b$.

2.2 Sketch Implementation

We want to find the highest positively and negatively correlated pairs over sliding window lengths. That could entail redoing the above operations every basic window. Because that could be expensive, SketchStream makes use of structured random vectors.

The idea is to form each structured random vector r from the concatenation of $nb = \frac{sw}{bw}$ random vectors $r = s_1, s_2, \dots, s_{nb}$ where each s_i has length bw . Furthermore, each s_i is either u or $-u$, and u is a random vector in the space $\{-1, +1\}^{bw}$ ¹. This choice is determined by another random binary k -vector b : if $b_i = 1 \Rightarrow s_i = u$ and if $b_i = 0 \Rightarrow s_i = -u$. The structured approach leads to an asymptotic performance of $O(nb)$ integer additions and $O(\log bw)$ floating point operations per time series, per random vector. There is a 30 to 40 factor improvement in runtime over recomputing the structured vectors from scratch.

In order to compute the dot products with structured random vectors, we first compute dot products with the random vector u . For each random vector r of length equal to the sliding window length $sw = nb * bw$, the dot product with each successive length sw chunk of the stream² is computed to form the sketch.

Example

The theory above may be somewhat difficult to understand at first, thus so consider this simple example.

As we mentioned, a random vector r_{bw} is constructed as follows:

$$r_{bw} = (r_0, r_1, \dots, r_{bw})$$

¹The random vector u remains the same for all s_i , only the sign changes.

²Successive chunks being one timepoint apart and bw being the length of a basic window.

where

$$\{r_i\} = \begin{cases} +1, & \text{with probability } \frac{1}{2}, \\ -1, & \text{with probability } \frac{1}{2}. \end{cases}$$

To form a random vector of length sw , another random vector b is constructed of length $nb = \frac{sw}{bw}$, which we call the control vector, as follows:

$$b = (b_0, b_1, \dots, b_{nb})$$

where

$$\{b_i\} = \begin{cases} +1, & \text{with probability } \frac{1}{2}, \\ -1, & \text{with probability } \frac{1}{2}. \end{cases}$$

The random vector r for a sliding window is then built as follows:³

$$r = (r_{bw} * b_0, r_{bw} * b_1, \dots, r_{bw} * b_{nb})$$

Now let us define the following example. Let's assume we are given a time series $X = (x_1, x_2, \dots)$, sliding window of size $sw = 12$, and a basic window of size $bw = 4$. If the random vector within a basic window is $r_{bw} = (1, 1, -1, 1)$, the control vector $b = (1, -1, 1)$, the random vector for a sliding window will be $r_{sw} = (1, 1, -1, 1, -1, -1, 1, -1, 1, 1, -1, 1)$.

So now we will form the sketches out of the two streams (of sw size).

$$X_{sw}^1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}), \quad (1)$$

$$X_{sw}^5 = (x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}). \quad (2)$$

Sketch for X^1 is

$$\begin{aligned} X_{sk}^1 &= r_{sw} * X_{sw}^1 \\ &= b_1 * r_{bw} * (x_1, x_2, x_3, x_4) + b_2 * r_{bw} * (x_5, x_6, x_7, x_8) \\ &\quad + b_3 * r_{bw} * (x_9, x_{10}, x_{11}, x_{12}) \\ &= b * (r_{bw} * (x_1, x_2, x_3, x_4), r_{bw} * (x_5, x_6, x_7, x_8), \\ &\quad r_{bw} * (x_9, x_{10}, x_{11}, x_{12})). \end{aligned} \quad (3)$$

Sketch for X^5 is

$$\begin{aligned} X_{sk}^5 &= r_{sw} * X_{sw}^5 \\ &= b_1 * r_{bw} * (x_5, x_6, x_7, x_8) + b_2 * r_{bw} * (x_9, x_{10}, x_{11}, x_{12}) \\ &\quad + b_3 * r_{bw} * (x_{13}, x_{14}, x_{15}, x_{16}) = b * (r_{bw} * (x_5, x_6, x_7, x_8), \\ &\quad r_{bw} * (x_9, x_{10}, x_{11}, x_{12}), r_{bw} * (x_{13}, x_{14}, x_{15}, x_{16})). \end{aligned} \quad (4)$$

³This reduction of randomness does not noticeably diminish the accuracy of the sketch estimation.

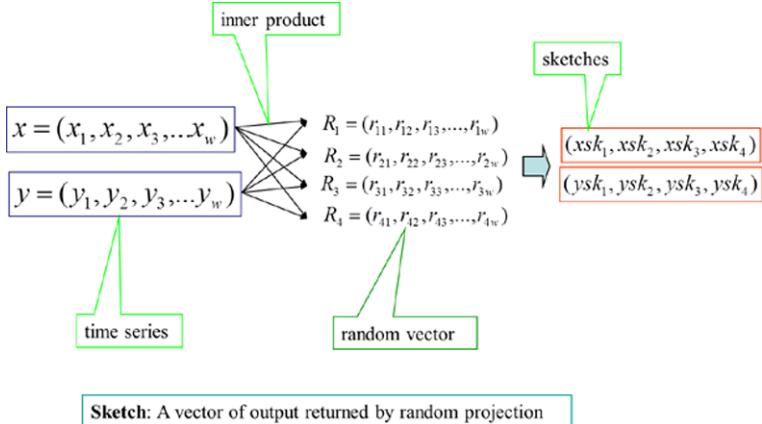


Fig. 3 Sketch by random projection

2.3 Sketch Vector Partitioning

Suppose we have 60^4 random vectors to which each window is compared and the sketch vector is the vector of the dot products to those random vectors. In Fig. 3, we can graphically see the procedure described above.

2.4 Grid Structure

To determine the closeness of the vectors, we partition each sketch vector into sub-vectors and build data structures to determine the similarity of the sub-vectors.

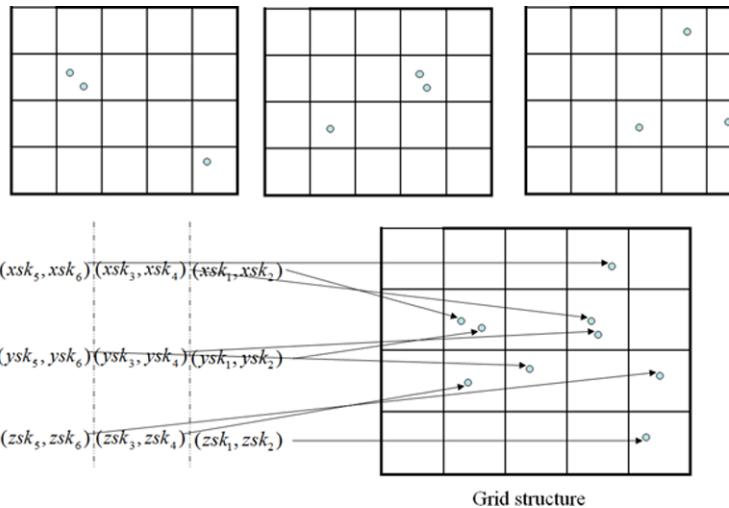
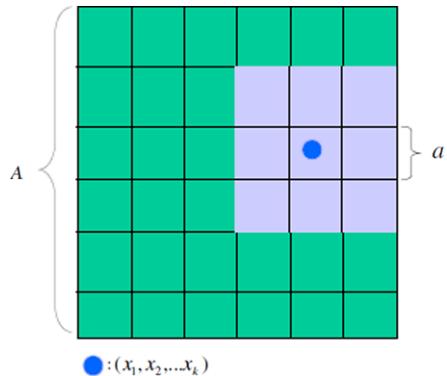
So the algorithmic framework described above, and presented in the papers related to StatStream [2, 4, 16], behaves as follows. Suppose we are seeking points within some distance d in the original time series space.

- Partition each sketch vector s of size N into groups of some size g .
- The i th group of each sketch vector s is placed in the i th grid structure of dimension g (in Fig. 5, $g = 2$).
- If two sketch vectors s_1 and s_2 are within distance $c * d^5$ in more than a fraction f of the groups, then the corresponding windows are candidate-highly-correlated windows and will be checked exactly.

In Fig. 4, you can see the how the sketches are compared and the grid filtering outputs the candidate highly correlated pairs.

⁴This is a parameter that can be modified.

⁵ c is a user-defined parameter.

**Fig. 4** Grid structure**Fig. 5** Grid structure

In order to derive a better intuition about the grid structure of Fig. 5, an analysis of its components is desirable. Assume a set of data points in a 2D space, where a 2-dimensional orthogonal regular grid is super-imposed on this space. In practice, the indexed space is bounded. Let the spacing of the grid be a . The indexing space, a 2-dimensional square with diameter A and partitioned into $\lceil \frac{A}{a} \rceil^2$ small cells. Each cell is a 2-dimensional square with side length a . All the cells are stored in a 2-dimensional array in main memory.

In such a main memory grid structure, we can compute the cell to which a particular point belongs. Let us use (c_1, c_2) to denote a cell that is the c_1 th in the first dimension and the c_2 th in the second dimension. A point p with coordinates x_1, x_2 is within the cell $(\lfloor \frac{x_1}{a} \rfloor, \lfloor \frac{x_2}{a} \rfloor)$. We say that point p is mapped to that cell. This can be easily extended to k -dimensions.

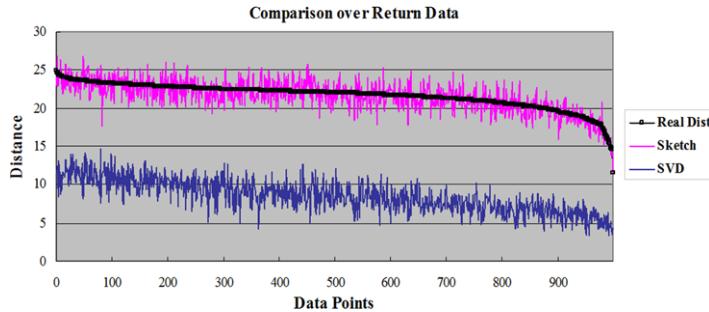


Fig. 6 Sketch by random projection

2.5 Results

Sketches work very well for uncooperative data. Figure 6 compares the distances of the Fourier and sketch approximations for 1000 pairs of 256 timepoint windows with a basic window size of length 32. Here sketch size = 30 and Singular Value Decomposition coefficient number = 30. As one can see, the sketch distances are closer to the real distance.

3 LearnStream

After finding the highest positively and negatively correlated time series to our target using SketchStream, we use stochastic gradient descent to predict complex linear relationships on streams. The LearnStream platform is thus capable of combining stream processing and machine learning.

3.1 Sliding Window Stochastic Gradient Descent

To evaluate a classifier's performance, given by a machine learning scheme, either a special testing set or a cross-validation technique may be employed. A test set contains pre-select examples different from those in the training set, and is used only for evaluation, not for training.

If data are scarce, it is sensible to use cross-validation in order not to waste any data, which could be useful in enhancing classifier performance; all data are used both for training the classifier and for testing its performance.

More examples do not necessarily mean better classifier performance. Even though the classifier becomes better in the training set it could actually perform worse on the test data. This is due to the possibility of overfitting of the classifier function, so that it fits too tightly to the training data. As an extreme example,

suppose that we want to predict the wealth of a person and one of the attributes is a unique key social security number. Overfitting might lead us to conclude that the most informative attribute is the social security number because given that, we would (in our training set) know the person's wealth. In Sect. 7, we describe how we found the most appropriate time window for each application.

As with every other classifier, Stochastic Gradient Descent has to be fitted with two arrays: an array X of size $[n_{samples}, n_{features}]$ holding the training samples, and an array y of size $n_{samples}$ holding the target values (class labels) for the training samples.

We train our regressor over a sliding window $[t_i, t_{i+sw}]$ and then use the regressor model obtained to predict the value for t_{i+sw+1} . For our predictions to have meaning, we need to use the data up to time point t to predict values for $t + 1$.

For example, we monitor the ticks of the prices for three stocks $A[t_0, t_{sw}]$, $B[t_0, t_{sw}]$ and $C[t_0, t_{sw}]$ and we want to make predictions for the price of $C[t_{sw+1}]$. To put this in the X - y framework, all the stocks in the X will start from time point t_0 and the target stock in y from t_1, \dots, t_n . This way we can train the model based on the relationship between the present and the future data.

3.2 Iterative Tuning of Sliding Window

We expect different outcomes with different window sizes. If the algorithm trains on 50 time points, the linear model will be different from when it trains on 5000.

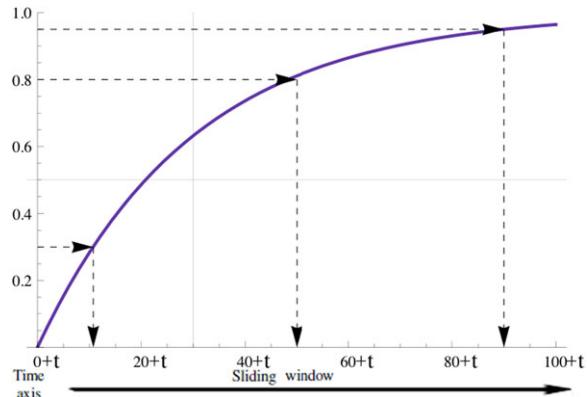
We don't know a priori which sliding window length will be capable of predicting with the smallest error. Therefore, an iterative tuning process is performed off-line to determine what would be a suitable interval for this particular dataset.

In particular, when we have a plethora of data at our disposal, we execute parallel versions of our algorithm with various window sizes. For each one we calculate the error of the prediction (relative to the correct result) and then we pick the sliding window which minimizes this error. Our assumption is that for a specific dataset the variance of its values will not dramatically change in the future. Varying this might be necessary for your application.

3.3 Exponential Random Picking

In time series analysis, recent data is often more relevant than older data. For example, the present stock price encodes more recent information than do historical prices. Traders do not want to forget the past entirely but they aim to place emphasis on the present as the balances may have changed concerning old statistics.

Therefore, in every time window we perform bootstrapping in order to try to decrease the variance of our classifier, giving preference to more recent values. For

Fig. 7 Exponential selection

example if SGD was to train a sliding window of the following values: 1, 2, 3, 4, 5 our algorithm randomly forms a new dataset which may have this form 3, 1, 4, 5, 5.⁶

The random selection is based on an exponential distribution which selects values closer to current time points, with higher probability. Figure 7 shows the probability of every time point in a sliding window to be a candidate for the new training dataset.

3.4 Warm Start/Use Historical Predictions

We may wish to focus more on recent data but that does not mean we should neglect the past.

Specifically, we used the coefficients produced by the last execution in the previous sliding window. By doing this we affect the algorithm's initialization point. This method works surprisingly well; much better than plain Stochastic Gradient Descent. In retrospect, this was a reasonable approach because our data is sequential, so the warm start would possibly give a good head start in contrast to starting from no a priori information.

3.5 Learning Rate

Despite the abundance of methods for learning from examples, there are only a few that can be used effectively for online learning. Classic batch training algorithms cannot straightforwardly handle non-stationary data. As described in [21], there exists the problem of “catastrophic interference”, in which training with new examples interferes excessively with previously learned examples, leading to saturation and slow convergence. Variants stochastic gradient descent methods are proposed in

⁶We allow selecting the same value more than once.

[22], which mainly implement an adaptive learning rate for achieving online learning. In Sect. 7, we discuss how we selected the appropriate α for our application.

4 Experimental Results

In this section, we present the experimental results of our work with real data and discuss the purity and the associated attributes of the three algorithms we studied.

The experiments compared three versions of Stochastic Gradient Descent:

- Plain Stochastic Gradient Descent.
From here on we will refer to it as *plainSGD*.
- Stochastic Gradient Descent with random sample picking.
From here on we will refer to it as *approxSGD*.
- Stochastic Gradient Descent with random sample picking and warm start.
From here on we will refer to it as *approxWarmSGD*.

5 Datasets

The experiments were mainly conducted on using three datasets: two proprietary FOREX (foreign exchange) datasets, with which we tested our trading strategy, as well as one publicly available household electric power consumption dataset.

5.1 FOREX Data Sets

Our primary dataset was obtained from Capital K Partners. It is composed of 28 different currency pairs with ticks aggregated every 100 ms. There are in total 80 days of data, starting from 01/05/2012 until 09/08/2012. Every pair consists of 42 features (such as the best bid/ask price, best bid/ask volume, mid prices, bid/ask range, bid/ask slope, etc.).

For the sake of simplicity, we used only the bid/ask prices. The procedure we followed was to use all the pairs associated with EUR currency (i.e., EURCAD, EURAUD, EURGBP) to predict the price of EURUSD pair. In other words, we form the training matrix X by using the prices from all the currency pairs and the target array y , by using the prices from the EURUSD pair.

The problem is that we could not predict both bid and ask prices simultaneously as they depend on different attributes.⁷ There are two separate instances of our algorithm running in parallel: one associated with the bid price and one with the ask

⁷We predicted the *mid price* = $\frac{\text{bid}+\text{ask}}{2}$ but we couldn't use it effectively in our trading strategy later on.

price. In this way, we can combine the predictions and create a buy/sell signal for our trading system.

The second dataset on FOREX data was obtained by a hedge fund from Singapore. Its far more simplistic and it contains ticks aggregated every 5 minutes for a time period of around three months. Because 5 minutes is so long, it omits a lot of information, so we do not expect results consistent with the 100 millisecond data.

5.2 Individual Household Electric Power Consumption Data Set [1]

This dataset contains measurements of electric power consumption in one household with a one-minute sampling rate over a period of almost 4 years. Different electrical quantities and some sub-metering values are available. This dataset consist of roughly 2070000 values for 4 years (16/12/2006–26/11/2010).

Its features are:

1. **Date**—Date in format dd/mm/yyyy.
2. **Time**—time in format hh:mm:ss.
3. **Global Active Power**—Household global minute-averaged active power (in kilo-watt).
4. **Global Reactive Power**—Household global minute-averaged reactive power (in kilowatt).
5. **Voltage**—Minute-averaged voltage (in volt).
6. **Global Intensity**—Household global minute-averaged current intensity (in ampere).
7. **Sub Metering 1**—Energy sub-metering No. 1 (in watt-hour of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave (hot plates are not electric but gas powered).
8. **Sub Metering 2**—Energy sub-metering No. 2 (in watt-hour of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.
9. **Sub Metering 3**—Energy sub-metering No. 3 (in watt-hour of active energy). It corresponds to an electric water-heater and an air-conditioner.

The reason we used a dataset so unrelated to the other two, which have an obvious financial similarity, was to cross-check the predictive accuracy of our algorithm and show that this work can be useful to fields other than financial analysis.

We use attributes 1, 2, 4, 5, 6, 7, 8, 9 to predict the global active power of the house for the next minute, something which may help us optimize our electricity usage. The method is almost the same as the one we followed for the financial datasets but somewhat simpler, as we deal with only one predictor.

6 Metrics

In this section, we describe the error metrics we used in order to compare the performance of the algorithms we tested.

6.1 Absolute/Relative Error

Experimental and measurement errors always create uncertainty in the final data. No measurement of a physical quantity can be entirely accurate.

The *absolute error* in a measured quantity is the uncertainty in the quantity and has the same units as the quantity itself.

For example, if you predict the price of a stock to be 1.3223 \$ \pm 0.00002 \$, the 0.00002 \$ is an absolute error.

The relative error (also called the fractional error) is obtained by dividing the absolute error in the quantity by the quantity itself:

$$\text{relative error} = \frac{\text{absolute error}}{\text{value of thing measured}},$$

or in terms common to Error Propagation

$$\text{relative error} = \frac{\Delta x}{x},$$

where x is any variable.

The relative error is usually more relevant than the absolute error. For example, a \$1 error in the price of a cheap stock is probably more serious than a \$1 error in the quote of an expensive one. Relative errors are dimensionless. When reporting relative errors, it is usual to multiply the fractional error by 100 and report it as a percentage.

Relative Error was our baseline method to determine whether or not our predictions were meaningful. Scikit-learn [17] implements three more metrics for regression, which we used to further analyze our estimator and are presented below.

6.2 Mean Squared Error

If \hat{y}_i is the predicted value of the i th sample and y_i is the corresponding true value, then the mean squared error (MSE) estimated over $n_{samples}$ is defined as:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} (y_i - \hat{y}_i)^2.$$

The best value is 0.0, higher values are worse.

6.3 Explained Variance Score

Explained variation measures the proportion to which a mathematical model accounts for the variation (dispersion) of a given data set.

If \hat{y} is the estimated target output and y is the corresponding (correct) target output, then the explained variance is estimated as follows:

$$\text{Explained Variance}(y, \hat{y}) = 1 - \frac{\text{Var}\{y - \hat{y}\}}{\text{Var}\{y\}}.$$

The best possible score is 1.0, lower values are worse.

6.4 R^2 Score, the Coefficient of Determination

The coefficient of determination, denoted R^2 , is used in the context of statistical models whose main purpose is the prediction of future outcomes on the basis of other related information.

R^2 is a number between 0 and 1.0, used to describe how well a regression line fits a set of data. An R^2 near 1.0 indicates that a regression line fits the data well, while an R^2 closer to 0 indicates a regression line does not fit the data very well. It is the proportion of variability in a data set that is accounted for by the statistical model. It provides a measure of how well *future outcomes* are likely to be predicted by the model.

If \hat{y}_i is the predicted value of the i th sample and y_i is the corresponding true value, then the score R^2 estimated over $n_{samples}$ is defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{samples}-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{samples}-1} (y_i - \bar{y})^2}$$

where $\bar{y} = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} y_i$.

The best possible score is 1.0, lower values are worse.

7 Parameter Tuning

As described in the previous sections, we have 5 distinct parameters in our algorithm. In order to determine the appropriate value for every attribute and for every dataset, we isolated them and through permutations, we modified only the parameter under consideration, so as to attain the optimal solution. In particular, in order to select the optimal parameter for each dataset, we performed grid-search on a set of possible value for each parameter and selected the set of parameters, for which the classifier that performs better.

Due to the fact that we base our model on an online framework, the boundaries between training and testing sort of overlap. In particular, our algorithm continuously updates the coefficients for every sliding window and thus learns and adapts to the future constantly. For our parameter tuning phase though, we used the first 80 % of our data to perform grid-search and fix the parameters. Then we kept them the same for the testing data.

- **Learning Rate (α) $\in \mathbb{R}$**

The learning rate can be any real value so it is not feasible to test every combination. We narrowed the problem down to determine how many decimal points will α have. As a matter of fact, we came to the realization that a value of two zeros after decimal point performs much better than a larger or a smaller learning rate. Reflecting that insight we set $\alpha = 0.0015$

- **Regularization Term $\in \{L1, L2, \text{Elastic Net}\}$**

$L1$ and $L2$ have almost the same results but for most of the cases $L2$ delivers slightly more accuracy. For the regularization coefficient C , which signifies the inverse of the regularization strength, we performed grid search for each application in the space of $[10^{-3}, 10^3]$ and select the best set of parameters.

- **Loss Function $\in \{\text{Squared Loss, Huber, Epsilon Insensitive}\}$**

Huber Loss and Squared Loss were much better on the average than Epsilon Insensitive Loss. Huber was slightly better than Squared loss for the *approxWarmSGD* for every dataset, while for the other two algorithms it was mainly dependent on the dataset.

In sum, the improvement or regression, in terms of relative error and the other methods stated, was sufficiently minor to avoid further examination.

- **Sliding Window (SW) $\in \mathbb{N}$**

The iterative tuning procedure followed in order to figure out the proper sw was explained in Sect. 3.2.

The results we obtained helped us set the

$$SW_{\text{capital K Forex Dataset}} = 72,$$

$$SW_{\text{singapore Forex Dataset}} = 100,$$

$$SW_{\text{Electrical Consumption Dataset}} = 155.$$

- **Approximation Time Points $\in SW_{size}$**

If the sliding window, on which plainSGD uses all the samples to train the model, is SW then let the size of the dataset used from *approxSGD* and *approxWarmSGD* be *approxSW*.

We found empirically that if we set $Size_{approxSW} = Size_{SW} - 1$, we produce a prediction with the smallest relative and mean squared error. We contrast this approach with setting $Size_{approxSW} = Size_{SW}/2$, in order to see how our algorithm behaves when we decrease the number of time points examined. These results could change in other data sets.

Table 1 Electrical consumption data $SW-1$ approx. Points

Metric	plainSGD	approxSGD	approxWarmSGD
Average relative error	46.5268326356 %	35.1390031025 %	12.3209254031 %
Mean squared error	1.20014251329	0.836286330532	0.15540327753
Explained variance	0.556287783866	0.654150947989	0.913688374021
R^2 score	0.333425379143	0.535515792892	0.913687016623

Table 2 Electrical consumption data $SW/2$ approx. Points

Metric	plainSGD	approxSGD	approxWarmSGD
Average relative error	46.5268326356 %	43.9709146395 %	12.3961653422 %
Mean squared error	1.20014251329	1.33460563753	0.154441109978
Explained variance	0.556287783866	0.550474129003	0.914222866339
R^2 score	0.333425379143	0.258742826809	0.914221416883

8 Experiments

8.1 Electric Power Consumption Data Set

In Tables 1 and 2 you see the results of the electric power consumption in terms of the metrics referred above. We present results both for $SW - 1$ and $\frac{SW}{2}$ approximation points. For this specific dataset the time lag we selected was 60, which means that we make our predictions for the electric power consumption for one hour ahead.

We can see that the method (*approxWarmSGD*) is better than the other two in all four metrics. The remarkable thing though is that even when we decrease the number of time points examined by half, there is almost no loss in the accuracy of the results.

8.2 Singapore Hedge Fund FOREX Data Set

Likewise, Tables 3 and 4 the results for the Singapore FOREX data set. The error metrics were almost the same for both bid and ask predictions so we show only one of them. As we described above, this dataset is not appropriate for high frequency trading, and we mainly use it in contrast with our second FOREX dataset that is high frequency and the accuracy there is far better. Nonetheless, the returns (even with a time lag of 5 minutes) are still bound to follow a random walk movement, thus we can classify them as uncooperative.

As we can see for this particular dataset, *approxSGD* without the warm start performs best. This reason may be that the values are infrequent (ticks every 5 minutes).

Table 3 Results for Singapore FOREX data *SW-1 approx. Points*

Metric	plainSGD	approxSGD	approxWarmSGD
	Bid/Ask	Bid/Ask	Bid/Ask
Average relative error	0.1366274551 %	0.133887583 %	0.2294997927 %
Mean squared error	5.79361776e-06	5.552743418e-06	1.547903339e-05
Explained variance	0.9990655232	0.9991046995	0.9974962369
R^2 score	0.9990535129	0.9990928638	0.9974712337

Table 4 Singapore FOREX data *SW/2 approx. Points*

Metric	plainSGD	approxSGD	approxWarmSGD
	Bid/Ask	Bid/Ask	Bid/Ask
Average relative error	0.1366274551 %	0.09281447603 %	0.1551851906 %
Mean squared error	5.79361778e-06	3.045854724e-06	7.963935037e-06
Explained variance	0.9990655232	0.9995082637	0.9987048952
R^2 score	0.9990535129	0.9995024072	0.9986989542

Table 5 Capital K FOREX data, *SW-1 approx. Points*

Metric	plainSGD	approxSGD	approxWarmSGD
	Bid/Ask	Bid/Ask	Bid/Ask
Average relative error	0.01167 %/0.0117 %	0.0116 %/0.0117 %	1.502e-03 %/1.554e-03 %
Mean squared error	2.201e-08/2.223e-08	2.175e-08/2.191e-08	1.287e-09/1.372e-09
Explained variance	0.999	0.999	0.999
R^2 score	0.9812/0.9828	0.9814/0.9831	0.9989/0.9989

8.3 Capital K FOREX Data Set

This brings us to the cornerstone of our data sets, the one on which we based our theoretical study to produce a trading strategy that could potentially generate substantial profit.

As stated above, we have 80 days of data each one of which consists of 860,000 ticks approximately. Each date is a separate dataset for which we have different results. In Tables 5 and 6, you can see the average values from all the dates.

approxWarmSGD is superior in all metrics. The results hold even if we use $\frac{SW}{2}$ time points.

Table 6 Capital K FOREX data, SW/2 approx. Points

Metric	plainSGD	approxSGD	approxWarmSGD
	Bid/Ask	Bid/Ask	Bid/Ask
Average relative error	0.01167 %/0.0117 %	0.0314 %/0.0313 %	1.701e-03 %/1.751e-03 %
Mean squared error	2.201e-08/2.223e-08	1.538e-07/1.517e-07	1.514e-09/1.602e-09
Explained variance	0.999	0.999	0.998
R^2 score	0.9812/0.9828	0.8687/0.8832	0.9987/0.9987

9 Trading Strategy

Most empirical studies with high-frequency data look at the time series of volatility, trading volume and spreads. Several researchers argue that all these time series follow a *U-shaped* or a *J-shaped* pattern, i.e., the highest point of these variables occurs at the opening of the trading day and they fall to lower levels during the mid-day period, and then rise again towards the close. The behavior of these variables is not easy to explain theoretically using the basic models related to three kinds of agents: the informed trader, the uninformed trader and the market maker.

The introduction of a distinction between discretionary and non-discretionary uninformed traders partially overcome this difficulty. If the uninformed or liquidity traders choose the time of their trades, then they can congregate in the periods when trading costs are low.

Some other models go further in explaining the positive relation between volatility and spread, indicating that more volatility is associated with the revelation of more information, and thus the market becomes more uncertain and spreads widen [23, 24].

9.1 Spread Based Trading

In contrast to the above, the approach we followed was again a very basic one, aimed at showing that the learning algorithm can make a constant positive profit even if the trading technique does not make use of macroeconomic theories or models.

Our strategy relies mostly on the spread of the bid and ask prices for consecutive time points. At this point we are not dealing with volatility; we trade one unit of EUR/USD per time. It is obvious though that in the case that you trade one unit and get a profit of 0.002 \$, it is the same as if you trade 1 million units and get a profit of 2000 \$. Of course, if you try to perform such a trade tracking volatility, you must check first that the amount of security you wish to buy is available.

Specifically, we have the two following cases where we get a buy/sell signal.

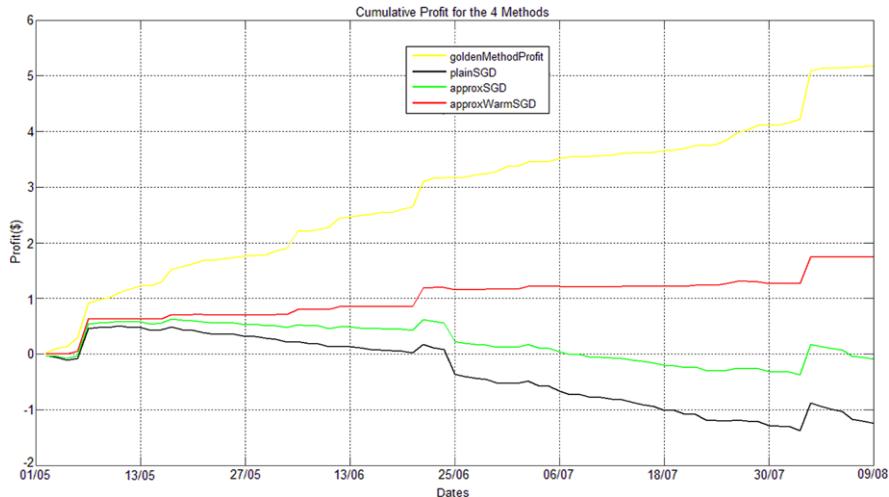


Fig. 8 Profit graph for every trading day

- If $bid_prediction_{t+1} > ask_t$ then buy at time t and sell at $t + 1$.
If the bid price at time point $t + 1$ is larger than the ask price at time t , it means that we will get a positive profit if we buy something that will be sold later for a higher price, than the price we bought it.
- If $ask_prediction_{t+1} < bid_t$ then sell at time t (sell short) and rebuy it at $t + 1$ to return to a neutral position.
If the ask price at time point $t + 1$ is smaller than the bid price at time t , it means that if we sell something now and rebuy it later we will gain a profit by exploiting this spread.

As is easily understood, though, if the prediction is not accurate we end up buying something that we have to sell for less or selling something and then need to buy it for a higher price.

Table 7 shows the trading results for every day and the total profit in the end. The first column (*goldenMethodProfit*), is the maximum profit we could gain if we predicted the exact value for the next time point. That is the unattainable ideal.

As we see *approxWarmSGD* is the only method that in the end has a positive profit. The astonishing thing is that even in the days when it loses, the algorithm somehow manages to limit the loss to about $\frac{1}{10}$ of the loss of the other two methods.

One can see the above results graphically represented in Fig. 8.

For the Singapore FOREX Dataset the results were not so encouraging though, as you can see in Table 8.

It is very difficult to capture the essence of the currency movement by using ticks aggregated every 5 minutes. At least, we can state that our algorithm could sustain a loss 37.14 times less than the other two algorithms.

Table 7 Capital K FOREX dataset trading results

Date	goldenMethodProfit	plainSGD	approxSGD	approxWarmSGD
2012/05/01	0.0239371 \$	-0.0342849 \$	-0.0269991 \$	-0.000753200000002 \$
2012/05/02	0.0761673 \$	-0.0291822 \$	-0.0175658 \$	0.00277210000001 \$
2012/05/03	0.0340376 \$	-0.0488826 \$	-0.0392353 \$	-0.0016023 \$
2012/05/04	0.1588481 \$	0.0274162 \$	0.0511831 \$	0.0477925 \$
2012/05/06	0.616827 \$	0.5445014 \$	0.5684931 \$	0.5750704 \$
2012/05/07	0.0641958 \$	0.0161211 \$	0.0215621 \$	0.0040079999999 \$
2012/05/08	0.0405388 \$	0.0038166999999 \$	0.0067438999999 \$	-0.0017237 \$
2012/05/09	0.089591 \$	0.0253449 \$	0.0266849 \$	0.0023354 \$
2012/05/10	0.0623359000001 \$	-0.0273292 \$	-0.0186682 \$	-0.0015336 \$
2012/05/11	0.0568553 \$	0.0016687 \$	0.0058607 \$	0.000805299999997 \$
2012/05/13	0.00609079999999 \$	-0.0541837 \$	-0.0421192 \$	-0.00012930000001 \$
2012/05/14	0.0702268 \$	0.0133073 \$	0.0181479 \$	-0.00031660000005 \$
2012/05/15	0.221093 \$	0.0479608 \$	0.0784407 \$	0.0801666 \$
2012/05/16	0.0533726 \$	-0.0497275 \$	-0.0294674 \$	-0.00666 \$
2012/05/17	0.0476873 \$	-0.0156991 \$	-0.0056944999999 \$	0.0122044 \$
2012/05/18	0.0625968 \$	-0.0339329 \$	-0.0171566 \$	-0.0048504000001 \$
2012/05/20	0.000837800000001 \$	-0.0306222 \$	-0.0220222 \$	-0.006 \$
2012/05/21	0.0275625 \$	0.0112561 \$	0.0120186 \$	-0.0011088 \$
2012/05/22	0.0213184 \$	-0.0146806 \$	-0.0121239 \$	-2.13000000002-05 \$
2012/05/25	0.0287465 \$	-0.0380112 \$	-0.0271716 \$	-0.0004648000001 \$
2012/05/27	0.00906550000004 \$	-0.0028059999998 \$	0.00326730000002 \$	0.0020467000001 \$
2012/05/28	0.0098993 \$	-0.0252188 \$	-0.0232469 \$	-0.0007262 \$
2012/05/29	0.0752841 \$	-0.0209611 \$	-0.002059000001 \$	0.0056653 \$

Table 7 (Continued)

Date	goldenMethodProfit	plainSGD	approxSGD	approxWarmSGD
2012/05/31	0.0482915 \$	-0.0511841 \$	-0.0312588 \$	0.00608160000001 \$
2012/06/01	0.3012095 \$	0.0108270999999 \$	0.0537842 \$	0.0870463 \$
2012/06/03	0.00031913 \$	-0.0326401 \$	-0.0230999 \$	0 \$
2012/06/04	0.0302697 \$	-0.00652450000001 \$	-0.00154900000001 \$	-0.000264900000003 \$
2012/06/05	0.0337185 \$	-0.0566453 \$	-0.0463728 \$	-0.0026809 \$
2012/06/06	0.1704434 \$	0.000312599999958 \$	0.0245615 \$	0.0551334 \$
2012/06/12	0.0159152 \$	-0.0018495 \$	-0.0022344 \$	-0.0003722 \$
2012/06/13	0.0279101 \$	-0.020317 \$	-0.0135648 \$	0.00335750000001 \$
2012/06/14	0.02000004 \$	-0.0257194 \$	-0.0146346 \$	-0.0013187 \$
2012/06/15	0.0307023 \$	-0.0091405 \$	-0.0005642 \$	0.00025649999997 \$
2012/06/17	0.00072620000001 \$	-0.015285 \$	-0.007924 \$	-0.00023 \$
2012/06/18	0.0066389499999 \$	-0.0067474 \$	-0.0041927 \$	-0.0027532 \$
2012/06/19	0.0407958 \$	-0.0320735 \$	-0.0163544 \$	-0.0030473 \$
2012/06/20	0.44752 \$	0.1504543 \$	0.1960734 \$	0.3364993 \$
2012/06/21	0.0622062999999 \$	-0.0545687000001 \$	-0.0378072000001 \$	0.0123713999999 \$
2012/06/22	0.0141182 \$	-0.0321446 \$	-0.0313011 \$	-0.0006107 \$
2012/06/24	7.3499999994e-05 \$	-0.4442654 \$	-0.3293038 \$	-0.04096 \$
2012/06/25	0.0091785 \$	-0.0430264 \$	-0.0358235 \$	0 \$
2012/06/26	0.0389425 \$	-0.0314376 \$	-0.0156091 \$	0.003335050000001 \$
2012/06/27	0.0181894 \$	-0.0221411 \$	-0.0136355 \$	0.0021179 \$
2012/06/28	0.0455772 \$	-0.0557239 \$	-0.0366159 \$	-0.0013171 \$
2012/06/29	0.0906558 \$	-0.0145692 \$	0.0014315 \$	0.0019478 \$
2012/07/01	0.000680899999999 \$	-0.000330600000002 \$	-0.000240600000001 \$	0 \$

Table 7 (Continued)

Date	goldenMethodProfit	plainSGD	approxSGD	approxWarmSGD
2012/07/02	0.0775112000001 \$	0.0452053000001 \$	0.0510538000001 \$	0.0553430000001 \$
2012/07/03	0.0058701 \$	-0.0830429 \$	-0.0662699 \$	0 \$
2012/07/04	0.0081751999999 \$	-0.0146892 \$	-0.0096125 \$	-6.6100000006e-05 \$
2012/07/05	0.0513885 \$	-0.0822023 \$	-0.059481 \$	-0.0063864 \$
2012/07/06	0.0273257 \$	-0.0662285 \$	-0.0511703 \$	-0.0009453 \$
2012/07/08	0.0020017 \$	0 \$	0 \$	0 \$
2012/07/09	0.00954100000001 \$	-0.0520138 \$	-0.043469 \$	-0.0004271 \$
2012/07/10	0.00594 \$	-0.00046 \$	-0.0007281 \$	-0.0002872 \$
2012/07/11	0.0164148 \$	-0.0272327 \$	-0.0189462 \$	-0.0023874 \$
2012/07/12	0.0290472 \$	-0.00936000000001 \$	-0.000822500000012 \$	0.00806419999999 \$
2012/07/13	0.0067269 \$	-0.0519178 \$	-0.0361188 \$	-0.0003176 \$
2012/07/15	1.91e-05 \$	-0.0423109 \$	-0.02321 \$	0 \$
2012/07/16	0.0119677 \$	-0.027802 \$	-0.0227166 \$	-0.0005198 \$
2012/07/17	0.0287962 \$	-0.0681972 \$	-0.044853 \$	0.0006499999996 \$
2012/07/18	0.0110806 \$	-0.0669917 \$	-0.0040249 \$	-0.0001258 \$
2012/07/19	0.028801 \$	-0.062382 \$	-0.0365526 \$	0.0017298 \$
2012/07/20	0.0542126 \$	-0.0070123 \$	0.0121815 \$	0.0173619 \$
2012/07/22	0.00097989999999 \$	-0.0967228 \$	-0.0575202 \$	0 \$
2012/07/23	0.0198586 \$	-0.0115944 \$	-0.010476 \$	-0.000934000000001 \$
2012/07/24	0.0824358999999 \$	-0.0155094000001 \$	0.0083618999994 \$	0.0312996 \$
2012/07/25	0.1288265 \$	0.0223879 \$	0.0381827 \$	0.0423119 \$
2012/07/26	0.0470796 \$	-0.0193145 \$	-0.0121306 \$	-0.0037587 \$
2012/07/27	0.0880507 \$	-0.00642770000001 \$	0.0040804999999 \$	-0.00492140000001 \$

Table 7 (Continued)

Date	goldenMethodProfit	plainSGD	approxSGD	approxWarmSGD
2012/07/29	8.77899999996e-05 \$	-0.07629 \$	-0.05687 \$	-0.031028 \$
2012/07/30	0.00886399999999 \$	-0.006927 \$	-0.0040943 \$	-0.0004847 \$
2012/07/31	0.0392846 \$	-0.0117898 \$	0.0002319 \$	0.0014711 \$
2012/08/01	0.0655613 \$	-0.0692096 \$	-0.056529 \$	-0.0046502 \$
2012/08/02	0.8581657 \$	0.4955924 \$	0.5432271 \$	0.4788215 \$
2012/08/03	0.0473335 \$	-0.059746 \$	-0.0366579 \$	0.0026634 \$
2012/08/05	0.00332218 \$	-0.0559779 \$	-0.0324349 \$	-0.0001001 \$
2012/08/06	0.0125088 \$	-0.0398368 \$	-0.0284713 \$	-0.0005952 \$
2012/08/07	0.0070426 \$	-0.1341935 \$	-0.1111658 \$	0 \$
2012/08/08	0.011554 \$	-0.0293867 \$	-0.0184645 \$	-8.99999999e-05 \$
2012/08/09	0.0214832 \$	-0.0484038 \$	-0.0326047 \$	0.0014498 \$
Total	5.1889791\$	-1.2490645 \$	-0.0894891 \$	1.7545607 \$

Table 8 Singapore FOREX dataset trading results

goldenMethodProfit	plainSGD	approxSGD	approxWarmSGD
11.810265 \$	-5.259669 \$	-5.209989 \$	-0.14704 \$

9.2 Other Methods we Tried

We thought of other techniques as well that take into account long term trading. Those methods though need further examination as problems could occur, for instance, in the case that we do not sell if the prediction is not right and just hold the amount purchased until we find a more profitable choice, we might end up stacking up our inventory and keep losing money as we do not sell anything.

Although it is not easy to develop such a strategy, we believe that we can generate much better results if we design a method that tries to minimize loss and not immediately buy and sell to return to a neutral position.

10 Conclusion

In this paper, we developed a framework called StatLearn consisting of fast algorithms for time series data streams. StatLearn concentrates on solving two problems: finding correlations in an online sliding window manner and predicting the future of financial time series based on an online approximation algorithm we created. We tested our results with data collected from the stock market and publicly available data sets.

10.1 Future Work

The work in this paper can be used as the basis for several future research directions including improving the trading strategy (especially the thresholds used to open or close a trade), incorporating other machine learning techniques such as boosted random forests, and finding outliers.

References

1. A. Frank, A. Asuncion, *UCI Machine Learning Repository* (2010). <http://archive.ics.uci.edu/ml>
2. R. Cole, D. Shasha, X. Zhao, Fast window correlations over uncooperative time series, in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD'05*, New York, NY, USA (ACM, New York, 2005), pp. 743–749
3. X. Zhao, High performance algorithms for multiple streaming time series. PhD thesis, New York, NY, USA, AAI3205697 (2006)

4. Y. Zhu, D. Shasha, Statstream: statistical monitoring of thousands of data streams in real time, in *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB'02, VLDB Endowment* (2002), pp. 358–369
5. R. Agrawal, C. Faloutsos, A.N. Swami, Efficient similarity search in sequence databases, in *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms, FODO'93*, London, UK, UK (Springer, Berlin, 1993), pp. 69–84
6. C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases. Technical report College Park, MD, USA (1993)
7. D. Rafiei, A. Mendelzon, Similarity-based queries for time series data. *SIGMOD Rec.* **26**(2), 13–25 (1997)
8. K.-P. Chan, A.W.-C. Fu, Efficient time series matching by wavelets, in *Proceedings of the 15th International Conference on Data Engineering, ICDE'99*, Washington, DC, USA (IEEE Comput. Soc., Los Alamitos, 1999), p. 126
9. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *VLDB* (2001), pp. 79–88
10. F. Korn, H.V. Jagadish, C. Faloutsos, Efficiently supporting ad hoc queries in large datasets of time sequences, in *SIGMOD* (1997), pp. 289–300
11. E. Keogh, K. Chakrabarti, M. Pazzani, S. Mehrotra, Dimensionality reduction for fast similarity search in large time series databases. *J. Knowl. Inf. Syst.* **3**, 263–286 (2000)
12. D. Achlioptas, *Database-Friendly Random Projections* (ACM, New York, 2001), pp. 274–281
13. W.B. Johnson, J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space (1982)
14. E. Kushilevitz, R. Ostrovsky, Y. Rabani, Efficient search for approximate nearest neighbor in high dimensional spaces, in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC'98*, New York, NY, USA (ACM, New York, 1998), pp. 614–623
15. P. Indyk, Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM* **53**(3), 307–323 (2006)
16. D. Shasha, Statstream pictorial architecture (2005). <http://cs.nyu.edu/shasha/papers/statstream/architecture.html>
17. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python. *J. Mach. Learn. Res.* **12**, 2825–2830 (2011)
18. M. Kontaki, A.N. Papadopoulos, Efficient similarity search in streaming time sequences, in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management, SSDBM'04*, Washington, DC, USA (IEEE Comput. Soc., Los Alamitos, 2004), p. 63
19. T. Buchgraber, D. Shutin, H.V. Poor, A sliding-window online fast variational sparse Bayesian learning algorithm, in *ICASSP* (2011), pp. 2128–2131
20. M. Velasco, A. Oliveira, C. Lyra, Neural networks give a warm start to linear optimization problems, in *Proceedings of the 2002 International Joint Conference Neural Networks, IJCNN'02*, vol. 2 (2002), pp. 1871–1876
21. R.S. Sutton, S.D. Whitehead, Online learning with random representations, in *Proceedings of the Tenth International Conference on Machine Learning* (Morgan Kaufmann, San Mateo, 1993), pp. 314–321
22. D. Saad (ed.), *On-Line Learning in Neural Networks* (Cambridge University Press, Cambridge, 1998)
23. C.C. Chang, S.S. Chen, R. Chou, C.W. Hsin, Intraday return spillovers and its variations across trading sessions. *Rev. Quant. Finance Account.* **36**, 355–390 (2011)
24. C.M.C. Lee, B. Mucklow, M.J. Ready, Spreads, depths, and the impact of earnings information: an intraday analysis. *Rev. Financ. Stud.* **6**(2), 345–374 (1993)

Adaptive, Automatic Stream Mining

Spiros Papadimitriou, Anthony Brockwell, and Christos Faloutsos

Sensor devices and embedded processors are becoming widespread, especially in measurement/monitoring applications. Their limited resources (CPU, memory and/or communication bandwidth and power) pose some interesting challenges. We need concise, expressive models to represent the important features of the data, and lend themselves to efficient estimation. In particular, under these severe constraints, we want models and estimation methods which (a) require little memory and a single pass over the data, (b) can adapt and handle arbitrary periodic components, and (c) can deal with various types of noise.

This material is based upon work supported by the National Science Foundation under Grants No. DMS-9819950 and IIS-0083148.

This material is based upon work supported by the National Science Foundation under Grants No. IIS-9817496, IIS-9988876, IIS-0083148, IIS-0113089, IIS-0209107 IIS-0205224 INT-0318547 SENSOR-0329549 EF-0331657IIS-0326322 by the Pennsylvania Infrastructure Technology Alliance (PITA) Grant No. 22-901-0001, and by the Defense Advanced Research Projects Agency under Contract No. N66001-00-1-8936. Additional funding was provided by donations from Intel, and by a gift from Northrop-Grumman Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, or other funding parties.

S. Papadimitriou (✉)

Business School, Rutgers University, New Brunswick, NJ, USA

e-mail: spapadim@business.rutgers.edu

A. Brockwell

Department of Statistics, Carnegie Mellon University, Pittsburgh, PA, USA

e-mail: abrock@stat.cmu.edu

C. Faloutsos

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

e-mail: christos@cs.cmu.edu

We propose AWSOM (Arbitrary Window Stream mOdeling Method), which allows sensors in remote or hostile environments to efficiently and effectively discover interesting patterns and trends. This can be done automatically, i.e., with no prior inspection of the data or any user intervention and expert tuning before or during data gathering. Our algorithms require limited resources and can thus be incorporated in sensors—possibly alongside a distributed query processing engine [6, 10, 26]. Updates are performed in constant time with respect to stream size, using logarithmic space. Existing forecasting methods (SARIMA, GARCH, etc.) or “traditional” Fourier and wavelet analysis fall short on one or more of these requirements. To the best of our knowledge, AWSOM is the first framework that combines all of the above characteristics.

Experiments on real and synthetic datasets demonstrate that AWSOM discovers meaningful patterns over long time periods. Thus, the patterns can also be used to make long-range forecasts, which are notoriously difficult to perform. In fact, AWSOM outperforms manually set up auto-regressive models, both in terms of long-term pattern detection and modeling, as well as by at least $10\times$ in resource consumption.

1 Introduction

Several applications produce huge amounts of data in the form of a semi-infinite stream of values [16, 18, 19, 21], typically samples or measurements. Formally, a stream is a discrete sequence of numbers $X_0, X_1, \dots, X_t, \dots$. Time sequences have attracted attention [7], for forecasting in financial, sales, environmental, ecological and biological time series, to mention a few. However, several new and exciting applications have recently become possible.

The emergence of cheap and small sensors has attracted significant attention. Sensors are small devices that gather measurements—for example, temperature readings, road traffic data, geological and astronomical observations, patient physiological data, etc. There are numerous, fascinating applications for such sensors and sensor networks, in fields such as health care and monitoring, industrial process control, civil infrastructure [9], road traffic safety and smart houses, to mention a few. Although current small sensor prototypes [23] have limited resources (512 bytes to 128 KB of storage), dime-sized devices with memory and processing power equivalent to a PDA are not far away. In fact, PDA-like devices with data gathering units are already being employed in some of the above applications. The goal in the next decade is single-chip computers with powerful processors and 2–10 GB [9] of non-volatile storage. Furthermore, embedded processors are becoming ubiquitous and their power has yet to be harnessed. A few examples of such applications are (a) intelligent (*active*) disks [30] that learn input traffic patterns and do appropriate prefetching and buffering, (b) intelligent routers that monitor data traffic and simplify network management.

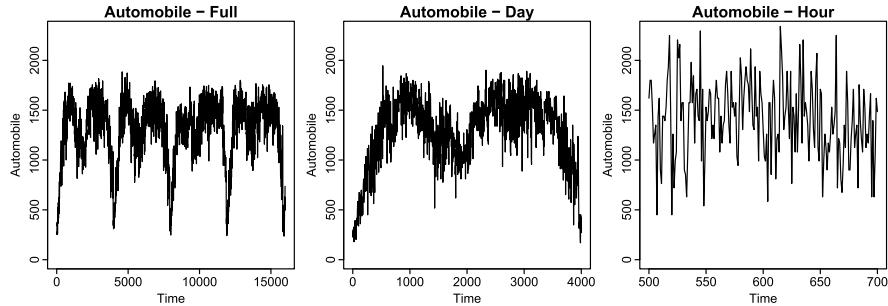


Fig. 1 Automobile traffic, complete series, one day and one hour (the *first two* are 8- and 4-point averages to make the trends easier to see). There is clearly a daily periodicity. Also, in each day there is another distinct pattern (morning and afternoon rush hours). However, at an hour scale traffic is highly bursty—in fact, it can be modeled by *self-similar* noise. We want a method that can capture all this information automatically, with one pass and using limited memory!

From now on, we use the term “sensor” broadly, to refer to any embedded computing device with fairly limited processing, memory and (optionally) communication resources and which generates a semi-infinite sequence of measurements.

The resource limitations unavoidably imply the need for certain trade-offs—it is impossible to store everything. Furthermore, we want to make the most of available resources, allowing the sensor to adapt and operate without supervision for as long as possible. This is the problem we address in this work. The goal is a “language” (i.e., model/representation) for efficient and effective stream mining. We want to collect information, in real-time and without any human intervention, and discover patterns such as those illustrated in Fig. 1.

This problem is orthogonal to that of continuous query processing. We focus on an adaptive algorithm that can look for arbitrary patterns and requires no prior knowledge and initial human tuning to guide it. There are situations when we do not know *beforehand* what we are looking for. Furthermore, it may be impossible to guide the sensor as it collects data, due to the large volume of data and/or limited or unavailable communication. If further exploration is desired, users can issue further queries, guided by the general long-term patterns to quickly narrow down the “search space.”

In detail, the main requirements are (see also Fig. 1):

1. *Concise models.* The models should be able to capture most of the regularities in real-world signals, using limited resources. In particular, we want
 - (a) Periodic component identification; humans can achieve this task, visually, from the time-plot. Our method should automatically spot multiple periodic components, each of unknown, arbitrary period.
 - (b) Noise filtering/identification; various types of “noise” are present in most real signals; our framework should deal with it and identify several of its characteristics.

- (c) Finally, we need a few, simple patterns (i.e., equations and features), so they can be easily communicated to other nearby sensors and/or to a central processing site.

2. *Streaming framework.* In particular, we want

- (a) An online, one-pass algorithm, since we can afford neither the memory nor time for offline updates, much less multiple passes over the data stream.
- (b) Limited memory use, since sensor memory will be exhausted, unless our method carefully detects redundancies (or equivalently, patterns) and exploits them.
- (c) Any-time forecasting and outlier detection; it is not enough to do compression (e.g., of long silence periods, or by ignoring small Fourier or wavelet coefficients). The model should be *generative* and thus able to report outliers. An outlier can be defined as any value that deviates too much from our forecast (e.g., by two standard deviations).

3. *Unsupervised, automatic operation.* In a general sensor setting, we cannot afford human intervention.

The AWSOM framework can extract several key features of the stream, in a principled way. It can do so in a single pass, with minimal resource requirements. Based on these features, it can immediately provide information about the stream at several levels. In brief, AWSOM has all of the above characteristics, while none of the previously published methods (AR and variations, Fourier analysis, wavelet decomposition—see Sect. 2.2) can claim the same.

2 Related Work

Previous work broadly falls into two categories. The first includes work done by the databases community on continuous query processing. These methods employ increasingly sophisticated mathematical methods, but the focus is typically on some form of compression (or, *synopses*) which do not employ generative models. The other includes various popular statistical models and methods for time series forecasting. However, standard estimation methods for these models typically fail in one or both of the requirements 2 and 3 earlier.

Thus the authors believe that there is a need for straightforward methods of time series model building which can be applied in real-time to semi-infinite streams of data, using limited memory. Table 1 summarizes the main characteristics of some key earlier techniques and our proposed AWSOM framework.

2.1 Continuous Queries and Stream Processing

An interesting method for discovering *representative trends* in time series using *sketches* was proposed by Indyk et al. [24]. A representative trend is a section of the

Table 1 Comparison of methods

Method	Contin. streams	Trends/Forecast	Automatic	Memory
DFT (N -point)	NO	NO	–	–
SWFT (N -point)	YES(?)	NO	–	–
DWT (N -point)	NO	NO	–	–
IncDWT [21]	YES	NO	–	–
Sketches [24]	NO	YES(?)	–	–
AR/ARIMA	YES	YES	NO [7]	W^2
AWSOM	YES	YES	YES	$m \mathcal{D} ^2$

time series that has the smallest sum of “distances” from *all* other sections of the same length. The proposed method employs random projections for dimensionality reduction and FFT to quickly compute the sum of distances. However, it cannot be applied to semi-infinite streams, since each section has to be compared to every other.

Gilbert et al. [21] use wavelets to compress the data into a fixed amount of memory, by keeping track of the largest Haar wavelet coefficients and carefully updating them online. In the following, we will use the name *Incremental DWT* or *IncDWT* for short. However, this method does not try to discover patterns and trends in the data. Thus, it cannot compete directly with our method, which employs a *generative* model. More recently, Garofalakis et al. [18] presented an approach for accurate data compression using *probabilistic wavelet synopses*. However, this method has an entirely different focus and cannot be applied to semi-infinite streams. The recent work of Guha et al. [22] efficiently constructs ϵ -accurate histograms in a stream setting, for a fixed number of buckets, using space and time-per-element that is logarithmic with respect to stream size (or poly-logarithmic with respect to a fixed window size). Further work on streams focuses on providing exact answers to pre-specified sets of queries using the minimum amount of memory possible. Arvind et al. [2] study the memory requirements of continuous queries over *relational* data streams. Datar et al. [14] keep *exact* summary statistics and provide theoretical bounds in the setting of a bit stream. Das et al. [13] examine the problem of discovering rules in time series, by quantizing them and then mining local, frequent-itemset type rules over sliding window fragments, based on certain similarity criteria.

There is also recent work on approximate answers to various types of continuous queries. Gehrke et al. [19] presents a comprehensive approach for answering correlated aggregate queries (e.g., “find points below the (current) average”), using histogram “summaries” to approximate aggregates. Dobra et al. [16] present a method for approximate answers to aggregate multi-join queries over several streams, using random projections and boosting. More recently, Considine et al. [12] have proposed novel sketching techniques for aggregate queries with the goal of minimizing communication and computation overhead.

Olston et al. [27] present a query processing framework for minimizing communication overhead over a set of specific continuous queries over multiple streams,

while providing error guarantees. Also, the work in [3] considers the problem of efficiently monitoring the top- k values in a distributed, multiple stream setting.

Zhu and Shasha [37] examine the problem of efficient detection of *elastic bursts* in streams. In [36], they use the DFT to summarize streams within a finite window and then compute the pairwise correlations among all streams. Also, Yi et al. [33] present a method for analysis of multiple co-evolving sequences.

A system for linear pattern discovery on multi-dimensional time series was presented recently by Chen et al. [11]. Although this framework employs varying resolutions *in time*, it does so by straight aggregation, using manually selected aggregation levels (although the authors discuss the use of a geometric progression of time frames) and can only deal with, essentially, linear trends. Recently, Bulut et al. [8] proposed an approach for hierarchical stream summarization (similar to that of [21]) which focuses on simple queries and communication/caching issues for wavelet coefficients. Even more recently, Palpanas et al. [28] consider approximation of time-series with *amnesic* functions. They propose novel techniques suitable for streaming, and applicable to a wide range of user-specified approximating functions.

Finally, in other areas of database research, Zhang et al. [35] present a framework for spatio-temporal joins using multiple-granularity indices. Aggregation levels are pre-specified and the main focus is on efficient indexing. Yufei et al. [31] recently presented an approach to recursively predict motion sequence patterns.

2.2 Time Series Methods

None of the continuous querying methods deal with pattern discovery and forecasting. The typical “textbook” approaches to forecasting (i.e., generative time series models) include *auto-regressive (AR)* models and their generalizations, *auto-regressive moving average (ARMA)*, *auto-regressive integrated moving average (ARIMA)* and *seasonal ARIMA (SARIMA)* [7]. Other popular time-series models include *GARCH (generalized auto-regressive conditional heteroskedasticity)* [5] and *ARFIMA (auto-regressive fractionally integrated moving average)* [4]. These time-series models could be used; however, standard estimation methods for these models fail in one or both of the requirements 2 and 3 stated in the introduction. Furthermore, these methods often have a number of other limitations.

Existing model-fitting methods are typically batch-based (i.e., do not allow online update of parameters). Established methods for determining model structure are at best computationally intensive, besides not easily automated. Large window sizes introduce severe estimation problems, both in terms of resource requirements as well as accuracy.

In addition, ARIMA models cannot handle bursty time series, even when the bursts are re-occurring. While GARCH models [5] can handle the class of bursty *white noise* sequences, they do not have the richness in structure to model a wide variety of different time series. Recently, the ARIMA model has been extended to

Table 2 Symbols and definitions

Symbol	Definition
$\mathbf{X}, \mathbf{P}, \dots$	Matrices (boldface capital).
$\mathbf{y}, \mathbf{q}, \mathbf{b}, \dots$	Vectors (boldface lower-case).
X_t	Value at time $t = 0, 1, \dots$ (sometimes also $X[t]$).
N	Number of points <i>so far</i> from $\{X_t\}$.
$W_{l,t}$	Wavelet (or, detail) coefficient (level l , time t); also denoted $W[l, t]$.
$V_{l,t}$	Scaling (or, smooth) coefficient (level l , time t); also denoted $V[l, t]$.
$\beta_i^{(l)}$	AWSOM coefficient at time lag i , for the equation at level l .
AWSOM(n_0)	AWSOM model of order n_0 (i.e., with n_0 model coefficients, per level). The <i>total order</i> of this model is $k \equiv n_0$.

ARFIMA, which handles the class of *self-similar* bursty sequences [4]. However, ARFIMA models pose particular computational problems since they cannot be expressed in Markovian form, making the likelihood computationally burdensome to evaluate.

All the above methods deal with linear forecasting. Nonlinear modeling methods [32] also require human intervention to choose the appropriate windows for non-linear regression or to configure an artificial neural network.

2.3 Other

There is a large body of work in the signal processing literature related to compression and feature extraction. Typical tools include the Fast Fourier Transform (FFT), as well as the Discrete Wavelet Transform (DWT) [29]. However, most of these algorithms (a) deal with *fixed length* signals of size N , and (b) cannot do forecasting (i.e., do not employ a *generative* model).

3 Background Material

In this section we give a very brief introduction to some necessary background material. Table 2 summarizes the key notation used in our development.

3.1 Auto-Regressive (AR) Modeling

Auto-regressive models are the most widely known and used. We present the basic ideas (see also Appendix A)—more information can be found in, e.g., [7]. The main

idea is to express X_t as a function of its previous values, plus (filtered) noise ϵ_t :

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_W X_{t-W} + \epsilon_t \quad (1)$$

where W is a window that is determined by trial and error, or by using a criterion that penalizes model complexity (i.e., large values of W), like the *Akaike Information Criterion (AIC)*. Seasonal variants (SAR, SAR(IMA)) also use window offsets that are multiples of a single, fixed period (i.e., besides terms of the form X_{t-i} , the equation contains terms of the form X_{t-Si} where S is a constant). The typical ARIMA modeling approach involves manual preprocessing to remove trend and seasonal components.

3.2 Recursive Least Squares (RLS)

Recursive Least Squares (RLS) is a method that allows dynamic update of a least-squares fit. The least squares solution to an overdetermined system of equations $\mathbf{X}\mathbf{b} = \mathbf{y}$ where $\mathbf{X} \in \mathbb{R}^{m \times k}$ (measurements), $\mathbf{y} \in \mathbb{R}^m$ (output variables) and $\mathbf{b} \in \mathbb{R}^k$ (regression coefficients to be estimated) is given by the solution of $\mathbf{X}^T \mathbf{X} \mathbf{b} = \mathbf{X}^T \mathbf{y}$. Thus, all we need for the solution are the projections

$$\mathbf{P} \equiv \mathbf{X}^T \mathbf{X} \quad \text{and} \quad \mathbf{q} \equiv \mathbf{X}^T \mathbf{y}.$$

We need only space $O(k^2 + k) = O(k^2)$ to keep the model up to date. When a new row $\mathbf{x}_{m+1} \in \mathbb{R}^k$ and output y_{m+1} arrive, we can update

$$\begin{aligned} \mathbf{P} &\leftarrow \mathbf{P} + \mathbf{x}_{m+1} \mathbf{x}_{m+1}^T \quad \text{and} \\ \mathbf{q} &\leftarrow \mathbf{q} + y_{m+1} \mathbf{x}_{m+1}. \end{aligned}$$

In fact, it is possible to update the regression coefficient vector \mathbf{b} without explicitly inverting \mathbf{P} to solve $P\mathbf{b} = \mathbf{P}^{-1}\mathbf{q}$. In particular (see, e.g., [34]) the update equations are

$$\begin{aligned} \mathbf{G} &\leftarrow \mathbf{G} - (1 + \mathbf{x}_{m+1}^T \mathbf{G} \mathbf{x}_{m+1})^{-1} \mathbf{G} \mathbf{x}_{m+1} \mathbf{x}_{m+1}^T \mathbf{G}, \\ \mathbf{b} &\leftarrow \mathbf{b} - \mathbf{G} \mathbf{x}_{m+1} (\mathbf{x}_{m+1}^T \mathbf{b} - y_{m+1}), \end{aligned}$$

where the matrix \mathbf{G} can be initialized to $\mathbf{G} \leftarrow \epsilon \mathbf{I}$ (where ϵ is a small positive number and \mathbf{I} is the $k \times k$ identity matrix).

RLS and AR

In the context of auto-regressive modeling (Eq. (1)), we have one equation for each stream value $X_{w+1}, \dots, X_t, \dots$, i.e., the m th row of the \mathbf{X} matrix above is

$$\mathbf{x}_m = [X_{m-1} \ X_{m-2} \ \dots \ X_{m-w}]^T \in \mathbb{R}^w$$

for $t - w = m = 1, 2, \dots$ ($t > w$). In this case, the solution vector \mathbf{b} consists precisely of the auto-regression coefficients in Eq. (1), i.e.,

$$\mathbf{b} = [\phi_1 \ \phi_2 \ \dots \ \phi_w]^T \in \mathbb{R}^w.$$

3.3 Wavelets

The N -point *discrete wavelet transform (DWT)* of a length N time sequence gives N wavelet coefficients. Wavelets are best introduced with the Haar transform because of its simplicity (a more rigorous introduction can be found, e.g., in [29]). At each level l of the construction we keep track of two sets of coefficients, each of which “looks” at a time window of size 2^l :

- $V_{l,t}$ —The *smooth* component, which consists of the $N/2^l$ *scaling coefficients*. These capture the low-frequency component of the signal; in particular, the frequency range $[0, 1/2^l]$.
- $W_{l,t}$ —The *detail* component, which consists of the $N/2^l$ *wavelet coefficients*. These capture the high-frequency component; in particular, the range $[1/2^l, 1/2^{l-1}]$.

The construction starts with $V_{0,t} = X_t$ and $W_{0,t}$ is not defined. At each iteration $l = 1, 2, \dots, \lg N$ we perform two operations on $V_{l-1,t}$ to compute the coefficients at the next level:

- Differencing, to extract the high frequencies:

$$W_{l,t} = (V_{l-1,2t} - V_{l-1,2t-1})/\sqrt{2};$$

- Smoothing, which averages¹ each consecutive pair of values and extracts the low frequencies:

$$V_{l,t} = (V_{l-1,2t} + V_{l-1,2t-1})/\sqrt{2}.$$

We stop when $W_{l,t}$ consists of one coefficient (which happens at $l = \lg N + 1$). The scaling coefficients are needed only during the intermediate stages of the computation. The final wavelet transform is the set of all wavelet coefficients along with $V_{\lg N + 1,0}$. Starting with $V_{\lg N + 1,0}$ (which is also referred to as the signal’s scaling coefficient) and following the inverse steps, we can reconstruct each $V_{l,t}$ until we reach $V_{0,t} \equiv X_t$.

Figure 2 illustrates the final effect for a signal with $N = 16$ values. Each wavelet coefficient is the result of projecting the original signal onto the corresponding basis signal (i.e., taking the dot product of the signal with the basis). Figure 2 shows the *scalogram*, that is, the energy (i.e., squared magnitude) of each wavelet coefficient

¹The scaling factor of $1/\sqrt{2}$ in both the difference and averaging operations is present in order to preserve total signal energy (i.e., sum of squares of all values).

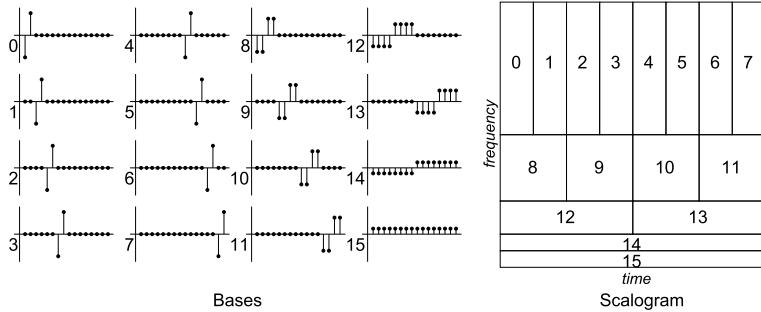


Fig. 2 Haar bases and correspondence to time/frequency (for signal length $N = 16$). Each wavelet coefficient is a linear projection of the signal to the respective basis

versus the location in time and frequency it is “responsible” for. In general, there are many wavelet transforms, but they all follow the pattern above: a wavelet transform uses a pair of filters, one high-pass and one low-pass. For example, in Haar wavelets, this pair consists of the simple differencing and averaging filters, respectively.

For our purposes here, we shall restrict ourselves to wavelets of the Daubechies family, which have desirable smoothness properties and successfully compress many real signals. In practice, although by far the most commonly used (largely due to their simplicity), Haar wavelets are too unsophisticated and introduce significant artifacts [29]. In fact, unless otherwise specified, we use Daubechies-6.

Incremental Wavelets

This part is a very brief overview of how to compute the DWT incrementally. This is the main idea of IncDWT [21], which uses Haar wavelets. In general, when using a wavelet filter of length L , the wavelet coefficient at a particular level is computed using the L corresponding scaling coefficients of the previous level. Recall that $L = 2$ for Haar (average and difference of two consecutive points), and $L = 6$ for Daubechies-6 that we typically use. Thus, we need to remember the last $L - 1$ scaling coefficients at each level. We call these the *wavelet crest*.

Definition 1 (Wavelet Crest) The *wavelet crest* at time t is defined as the set of scaling coefficients (wavelet smooths) that need to be kept in order to compute the new wavelet coefficients when X_t arrives.

Lemma 1 (DWT Update) *Updating the wavelet crest requires space $(L - 1) \lg N + L = O(L \lg N) = O(\lg N)$, where L is the width of the wavelet filter (fixed) and N the number of values seen so far.*

Proof See [21]. Generalizing to non-Haar wavelets and taking into account the wavelet filter width is straightforward. \square

Wavelet Properties

In this section, we emphasize the DWT properties which are relevant to AWSOM.

Computational Complexity

The DWT can be computed in $O(N)$ time and, as new points arrive, it can be updated in $O(1)$ amortized time. This is made possible by the structure of the time/frequency decomposition which is unique to wavelets. For instance, the Fourier transform also decomposes a signal into frequencies (i.e., sum of sines), but requires $O(N \lg N)$ time to compute and cannot be updated as new points arrive.

Time/Frequency Decomposition

Notice (see scalogram in Fig. 2) that higher level coefficients are highly localized in time, but involve uncertainty in frequency and vice-versa. This is a *fundamental trade-off* of any time/frequency representation and is a manifestation of the *uncertainty principle*, according to which localization in frequencies is inversely proportional to localization in time. The wavelet representation is an excellent choice when dealing with semi-infinite streams in limited memory: it “compresses” well many real signals, while it is fast to compute and can be updated online.

Wavelets and Decorrelation

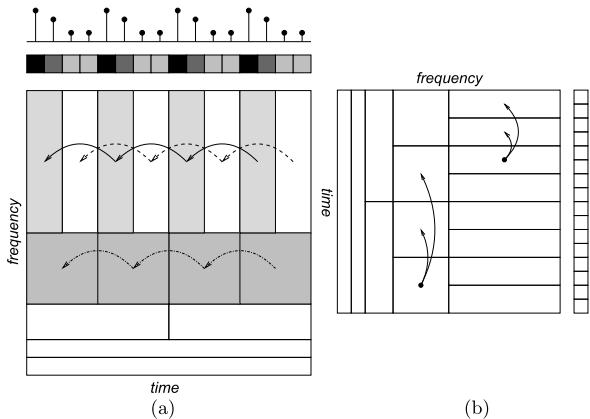
A wavelet transform with filter of length $2L$ can decorrelate only certain signals provided their L th order (or less) backward difference² is a stationary random process [29]. For real signals, this value of L is not known in advance and may be impractically large: the space complexity of computing new wavelet coefficients is $O(L \lg N)$ —see Lemma 1.

Wavelet Variance

One further benefit of using wavelets is that they decompose the variance across scales. Furthermore, the plot of log-power versus scale can be used to detect self-similar components (see Appendix B for a brief overview).

²In particular, the Daubechies- $2L$ wavelet filters are essentially L th order backward differences (and, consequently, the scaling filters are essentially $2L$ -point weighted moving averages).

Fig. 3 AWSOM—Intuition and demonstration. AWSOM captures intra-scale correlations (a). Also, (b) demonstrates why we fit different models per level



4 Proposed Method

In this section, we introduce our proposed model. What equations should we be looking for to replace ARIMA's (see Eq. (1))?

4.1 Intuition Behind Our Method

First Part—Information Representation

This is a crucial choice—what is a good way to represent the key information in the series, given the severe resource constraints in a streaming, sensor setting? We want a powerful and flexible representation that can adapt to the sequence, rather than expect someone to adapt the sequence to the representation. We propose to use wavelets because they are extremely successful in compressing most real signals, such as voice and images [17], seismic data [38], biomedical signals [1] and economic time sequences [20]. By using wavelet coefficients, we immediately discard many redundancies (i.e., near-zero valued wavelet coefficients) and focus on what really matters. Furthermore, the DWT can be computed quickly and updated online.

Second Part—Correlations

In the wavelet domain, how can we capture arbitrary periodicities? A periodic signal will have high-energy wavelet coefficients at the scales that correspond to its frequency. Also, successive coefficients on the same level should have related values (see Fig. 3(a)). Thus, in order to capture periodic components, we should look for intra-scale correlations between wavelet coefficients.

The last question we need to answer is: what type of regression models should we use to quantify these correlations? Our proposed method tries to capture these

by fitting linear regression models in the wavelet domain. These can also be updated online with RLS.

Summary

We propose using the wavelet representation of the series and capturing correlations in the wavelet domain (see Fig. 3(b)). If we naively try to apply linear auto-regression in the time domain, there are several problems:

- *Window size.* In order to capture long-term (non-sinusoidal) periodic components, the window size has to be as large as the period. If we hope to capture *any* information about long-term periodic trends, the window size has to be extremely large and this clearly violates the limited memory requirements (it also creates other problems, as discussed next).

However, the wavelet coefficients at each level capture information at a coarser resolution, but at windows whose size increases exponentially with the level.

- *Noise.* Even with a large window size, the presence of noise (typically at higher frequencies) severely affects model fitting. Dealing with noise in the time domain is possible (moving average being the simplest approach), but cannot be done easily in an online setting and often requires human intervention (seasonal models typically used for this reason, where the period has to be manually identified).

However, the wavelet transform filters out this noise quite successfully and does so in a principled manner, than has been proven to work for several real signals. Furthermore, it allows us to extract certain important characteristics of the noise.

Therefore, our approach significantly improves modeling power while dramatically reducing memory requirements, by essentially performing auto-regression on the wavelet representation instead of the original signal. At the same time, it adheres to the principle of no prior human intervention and tuning (or, adapting to new data as they arrive).

4.2 AWSOM Modeling

We express the wavelet coefficients at each level as a function of the n_0 previous coefficients of the same level, i.e.,

$$W_{l,t} = \beta_1^{(l)} W_{l,t-1} + \beta_2^{(l)} W_{l,t-2} + \cdots + \beta_{n_0}^{(l)} W_{l,t-n_0} \quad (2)$$

where $W_{l,t}$ are the wavelet coefficients (at time t and level l) and $\beta_i^{(l)}$ are the AWSOM coefficients (for level l and lag i). We estimate one set of such coefficients for *each* level l ; note that l is *at most* $\lg N$. This is a model of order n_0 , denoted as AWSOM(n_0). To summarize:

Input: Continuous, numerical stream $X[1], X[2], \dots, X[t], \dots$

Model order n_0 (typically $n_0 \leq 6$)

Output: Wavelet variances \mathcal{V}_l

AWSOM coefficients $\beta_i^{(l)}$

Wavelet crest (or, initial conditions) $W[N - i - 1, l]$

Where: N is the number of points seen so far

$1 \leq l \leq \lceil \lg N \rceil$ and $1 \leq i \leq n_0$

UpdateCrest ($X[t]$):

Foreach $l \geq 0$ s.t. 2^l divides t :

Compute $V[l, t/2^l]$

If 2^{l+1} divides t :

Compute $W[l, t/2^{l+1}]$

Delete $W[l, t/2^{l+1} - L]$

Update ($X[t]$):

UpdateCrest($X[t]$)

Foreach new coeff. $W[l, t']$ in wavelet crest:

Find the linear model it belongs to

based on l and $t' \bmod A$

Update $\mathbf{P} \equiv \mathbf{X}^T \mathbf{X}$ and

$\mathbf{q} \equiv \mathbf{X}^T \mathbf{y}$ for this model

ModelSelection:

For each linear model:

Estimate SSR of complete model

For each subset of regression variables:

Compute SSR of reduced model from Lemma 2

Estimate probability that reduction in variance is not due to chance

Select the subset of variables with highest probability (or keep all

if not within 95 % confidence interval)

Fig. 4 High-level description of the algorithms

Definition 2 (AWSOM Model and Order) A model with n_0 coefficients $\beta_i^{(l)}$, $i = 1, 2, \dots, n_0$ for each wavelet level l is denoted as AWSOM(n_0). Its (*total*) *order* is n_0 .

This can capture arbitrary periodic components and is sufficient in many real signals.

Figure 4 described the algorithm; we update the wavelet crest online (as described before) and use recursive least squares to update the regression models, also online.

4.3 Model Selection

Many of the dependencies may be statistically insignificant, so the respective coefficients $\beta_i^{(l)}$ should be set to zero. We want to (a) avoid over-fitting and (b) present to the user those patterns that are important. We can do model selection using *only* data gathered online and with time complexity *independent* of the stream size. Next, we show how feature selection can be done from the data gathered online (i.e., P and \mathbf{q} for each AWSOM equation). The algorithm is sketched in Fig. 4. The main idea

is to determine whether the reduction in error achieved by adding extra parameters is statistically significant (based on some criterion), i.e., it cannot be attributed to noise.

Model Testing and Selection

The key quantity we need is the total squared error (or, *square sum of residuals (SSR)*).

Lemma 2 (Square Sum of Residuals) *If \mathbf{b} is the least-squares solution to the overdetermined equation $\mathbf{X}\mathbf{b} = \mathbf{y}$, then*

$$s_n \equiv \sum_{i=1}^n (\mathbf{x}_i^T \mathbf{b} - y_i)^2 = \mathbf{b}^T \mathbf{P} \mathbf{b} - 2\mathbf{b}^T \mathbf{q} + \mathbf{y}^2.$$

Proof Straightforward from the definition of s_n , which in matrix form is $s_n = (\mathbf{X}\mathbf{b} - \mathbf{y})^2$. \square

Thus, besides \mathbf{P} and \mathbf{q} , we only need to update \mathbf{y}^2 (a single number), by adding y_i^2 to it as each new value arrives. Now, if we select a subset $\mathcal{I} = \{i_1, i_2, \dots, i_p\} \subseteq \{1, 2, \dots, k\}$ of the k variables x_1, x_2, \dots, x_k , then the solution $\mathbf{b}_{\mathcal{I}}$ for this subset is given by $\mathbf{P}_{\mathcal{I}} \mathbf{b}_{\mathcal{I}} = \mathbf{q}_{\mathcal{I}}$ and the SSR by $s_n = \mathbf{b}_{\mathcal{I}}^T \mathbf{P}_{\mathcal{I}} \mathbf{b}_{\mathcal{I}} - 2\mathbf{b}_{\mathcal{I}}^T \mathbf{q}_{\mathcal{I}} + \mathbf{y}^2$ where the subscript \mathcal{I} denotes straight row/column selection (e.g., $\mathbf{P}_{\mathcal{I}} = [p_{i_j, i_k}]_{i_j, i_k \in \mathcal{I}}$)

The *F-test (Fisher test)* [15] is a standard method for determining whether a reduction in variance is statistically significant. The F-test is based on the sample variances, which can be computed *directly* from the SSR (Lemma 2). Although the F-test holds precisely (i.e., non-asymptotically) under normality assumptions, in practice it works well in several circumstances, especially when the population size is large (as is the case with semi-infinite streams).

4.4 Complexity

In this section, we show that our proposed AWSOM models can be easily estimated with a single-pass, “any-time” algorithm. From Lemma 1, estimating the new wavelet coefficients requires space $O(\lg N)$. In fact, since we typically use Daubechies-6 wavelets ($L = 6$), we need to keep exactly $5\lg N + 6$ values. The AWSOM models can be dynamically updated using RLS.

At each level, we fit a separate regression model; since the number of levels is $\lceil \lg N \rceil$, this leads to the following result:

Lemma 3 (Logarithmic Space Complexity) *Maintaining an AWSOM(k) model requires $O(\lg N + mk^2)$ space, where N is the length of the signal so far, k is the total AWSOM order and $m = \lceil \lg N \rceil = O(\lg N)$ the number of equations.*

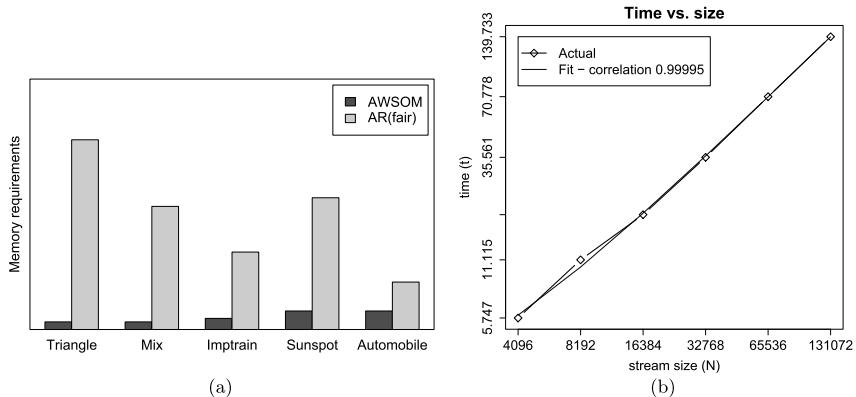


Fig. 5 (a) Memory space requirements (normalized): Space needed to keep the models up-to-date (AWSOM and AR with equivalent, fair window size). (b) Time complexity versus stream size (Python prototype), including model selection; the relationship is exactly linear, as expected

Proof Keeping the wavelet crest scaling coefficients requires space $O(\lg N)$. If we use recursive least squares, we need to maintain a $k \times k$ matrix for each of the m equations in the model. \square

Auto-regressive models with a comparable window size need space $O(m^2k^2)$, since the equivalent fair window size is $W \approx mk$. Here, “fair” means that the number of total number of AWSOM coefficients plus the number of initial conditions we need to store is the same for both methods. This is the information that comprises the data synopsis and that would have to be eventually communicated. However, the device gathering the measurements needs extra storage space in order to update the models. The latter is, in fact, *much* larger for AR than for AWSOM (see Fig. 5(a)). Thus this definition of equivalent window actually favors AR.

Lemma 4 (Time Complexity) *Updating the model when a new data point arrives requires $O(k^2)$ time on average, where k is the number of AWSOM coefficients in each equation.*

Proof On average, the wavelet crest scaling coefficients can be updated in $O(1)$ amortized time. Although a single step may require $O(\lg N)$ time in the worst case, on average, the (amortized) time required is $O(\sum_{i=0}^n \mathcal{B}(i)/N) = O(1)$ (where $\mathcal{B}(i)$ is the number of trailing zeros in the binary representation of i).³ Updating the $k \times k$ matrix for the appropriate linear equation (which can be identified in $O(1)$ time, based on the level l), requires time $O(k^2)$. \square

Auto-regressive models with a comparable window size need $O(m^2k^2)$ time per update.

³Seen differently, *IncDWT* is essentially a pre-order traversal of the wavelet coefficient tree.

Table 3 Description of datasets (sizes are in number of points, 1K = 1024 points)

Dataset	Size	Description
Triangle	64K	Triangle wave (i.e., piecewise linear; amplitude 5, period 256)
Mix	256K	Square wave (amplitude 25, period 256) plus sine (amplitude 5, period 64)
Sunspot	2K	Sunspot data
Automobile	32K	Automobile traffic sensor trace from large Midwestern state
Temperature	32K	Measurements from indoor temperature sensor (per second, degrees Celsius)

Corollary 1 (Constant-Time Update) *When the model parameters have been fixed (typically k is a small constant ≈ 6 and $m \sim \lg N$), the model requires space $O(\lg N)$ and amortized time $O(1)$ for each update.*

Figure 5(b) shows that this is clearly the case, as expected.

5 Experimental Evaluation

We compared AWSOM against standard AR (with the equivalent, fair window size—see Sect. 4.4), as well as hand-tuned (S)ARIMA (wherever possible). Our prototype AWSOM implementation is written in Python, using Numeric Python for fast array manipulation. We used the standard `ts` package from R⁴ for AR and (S)ARIMA models. We illustrate the properties of AWSOM and how to interpret the models using synthetic datasets and then show how these apply to real datasets (see Table 3).

Only the first half of each sequence was used to estimate the models, which were then applied to generate a sequence of length equal to that of the *entire* second half. For AR and (S)ARIMA, the last values (as dictated by the lags) of the first half were used to initiate generation. For AWSOM we again used as many of the last wavelet coefficients from each DWT level of the first half as were necessary to start applying the model equations. We should note that generating more than, say, 10 steps ahead is very rare: most methods in the literature [32] generate one step ahead, then obtain the correct value of X_{t+1} , and only then try to generate X_{t+2} . Nevertheless, our goal is to capture long-term behavior under severe resource constraints and AWSOM achieves this efficiently.

⁴R version 1.6.0; see <http://www.r-project.org/>.

5.1 Interpreting the Models

Visual Inspection

A “forecast” is essentially a by-product of any *generative* time series model: application of any model to generate a number of “future” values reveals precisely the trends and patterns captured by that model. In other words, synthesizing points based on the model is the simplest way for any user to get a quick, yet fairly accurate idea of what the trends are or, more precisely, what the *model* thinks they are. Thus, what we expect to see (especially in a long-range forecast) is the *important* patterns that can be identified from the real data. However, an expert user can extract even more precise information from the models.

Variance Test

As explained in Appendix B, if the signal is self-similar, then the plot of log-power versus scale is linear.

Definition 3 (Variance Log-Power Diagnostic) The log-power vs. scale plot is the *wavelet variance log-power diagnostic plot* (or just *log-power diagnostic*). In particular, the correlation coefficient ρ_α quantifies the relation. If the plot is linear (in a range of scales), the slope $\hat{\alpha}$ is the *self-similarity exponent* ($-1 < \alpha < 0$, closer to zero the more bursty the series).

A large value of $|\rho_\alpha|$, at least across several scales, indicates that the series component in those scales may be modeled using, e.g., a fractional noise process with parameter dictated by α (see `Automobile`). However, we should otherwise be careful in drawing further conclusions about the behavior within these scales.

We should note that after the observation by [25], fractional noise processes and, in general, self-similar sequences have revolutionized network traffic modeling. Furthermore, self-similar sequences appear in atomic clock fluctuations, river minima, compressed video bit-rates [4, 29], to mention a few examples.

Wavelet Variance (Energy and Power)

The magnitude of variance within each scale serves as an indicator about which frequency components are the dominant ones in the sequence. To precisely interpret the results, we also need to take into account the fundamental uncertainty in frequencies (see Fig. 12). However, the wavelet variance plot quickly gives us the general picture of important trends. Furthermore, it guides us to focus on AWSOM coefficients around frequencies with large variance.

To summarize, the steps are: (i) Examine the log-power diagnostic to identify sub-bands that correspond to a self-similar component. These may be modeled using

a fractional noise process for generation purposes; for forecasting purposes they are just that: noise. (ii) Examine the wavelet energy spectrum to quickly identify important sub-bands.

Experimental Goals

In each case, we demonstrate that AWSOM can provide information to answer the following questions, using *limited resources* and no supervision:

- (Q1) Identify and capture periodic components: this can be done by simply inspecting the forecasts, or by examining the wavelet variance. The latter also gives information about the relative “significance” (essentially, amplitude) of each component.
- (Q2) Diagnose the presence of self-similar noise in the appropriate scales: The log-power diagnostic provides the necessary information.
- (Q3) Perform long-range forecasts: Besides identifying periodic components, the AWSOM coefficients at the appropriate scales capture their behavior (*regardless* of their relative amplitude).

5.2 Synthetic Datasets

We present synthetic datasets to illustrate the basic properties of AWSOM, its behavior on several characteristic classes of sequences, and the principles behind interpreting the models. Applying the models to generate a number of “future” data points is the quickest way to see if each method captures long-term patterns (see Fig. 7).

Triangle

AR fails to capture anything because the window is not large enough. SAR estimation (with no differencing, no MA component and only a manually pre-specified 256-lag seasonal component) fails completely. In fact, R segfaults after several minutes, even without using maximum-likelihood estimation (MLE). However, AWSOM captures the periodicity.

This is immediately evident from the forecasts, which capture the trend almost perfectly. Also, inspection the wavelet variance (Fig. 9(a)) shows a single spike at scale 7 (which corresponds to a window of $2^7 = 128$; this is as expected, since there is zero change among consecutive windows of the next size, 256).

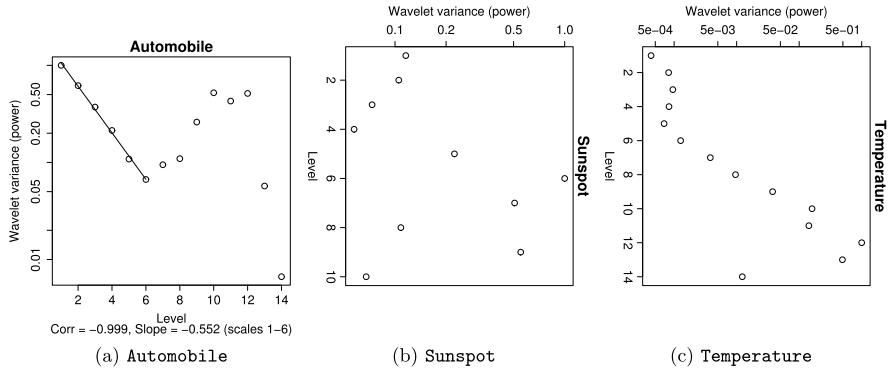


Fig. 6 Wavelet log-power diagnostic (real datasets). Horizontal axis is wavelet scale (l) and vertical axis is log-power ($\log V_l$) (see Appendix B). A linear trend with negative slope (between 0 and -1) indicates presence of self-similar noise. Automobile exhibits self-similarity in scales up to 6 (which roughly corresponds to one hour) but not overall

Mix

AR is again confused and does not capture even the sinusoidal component. SAR estimation (without MLE) fails (R's optimizer returns an error, after several minutes of computation).

Quick inspection of the AWSOM forecast shows clearly the two periodic components. The wavelet variance plot (Fig. 9(b)) also gives this information: there is again a spike at scale 7 (or, window $2^7 = 128$; same interpretation as Triangle), which corresponds to the square wave periodic component. There is also a rise at scale 5 (or window $2^5 = 32$), which corresponds to the sinusoidal periodic component⁵. The difference in magnitude of the variances is precisely due to the difference in amplitude of the two periodic components: indeed, the square wave component is a much “stronger” one. However, regardless of the strength (i.e., variance) of each component, the AWSOM coefficients capture their behavior. In summary, using limited resources, AWSOM provides all the key information about the series.

5.3 Real Datasets

For the real datasets, we again show long-range forecasts (see Fig. 8), as well as the marginal distribution *quantile-quantile plots* (or *Q-Q plots*—see Figs. 11 and 10).⁶

⁵The nonzero value at scale 6 is expected and due to frequency leaks, as explained in Appendix B.

⁶These are the scatter plots of (x, y) such that $p\%$ of the values are below x in the real sequence and below y in the generated sequence. When the distributions are identical, the Q-Q plot coincides with the bisector of the first quadrant.

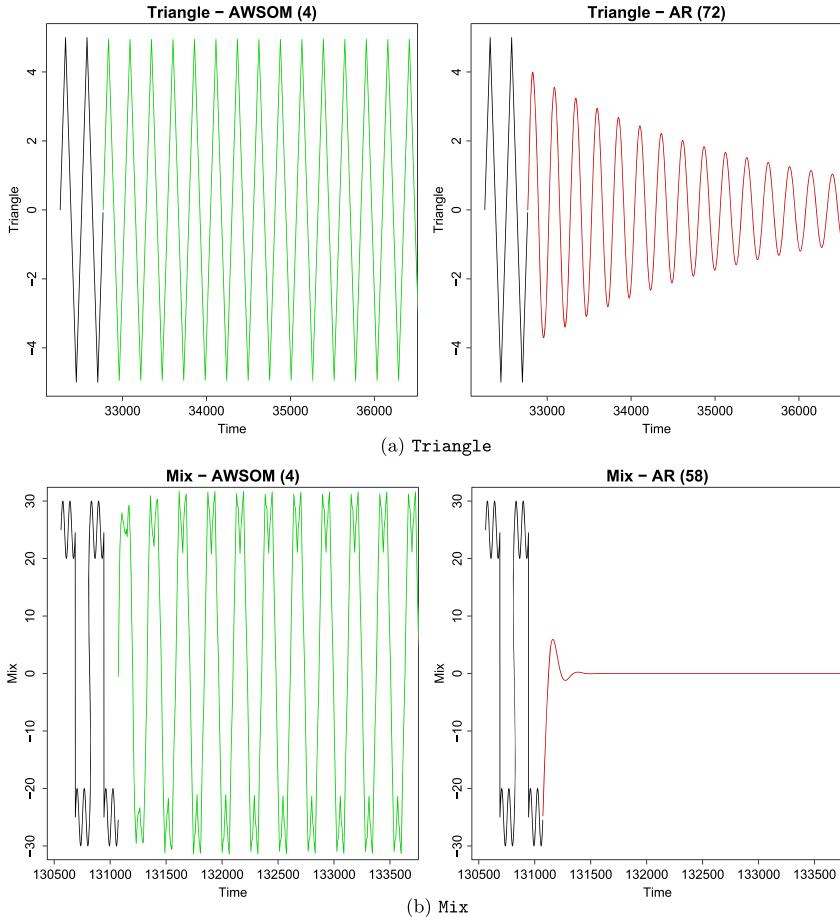


Fig. 7 Forecasts—synthetic datasets. Note that AR gives the wrong trend (if any), while seasonal AR fails to complete and is not shown

Sunspot

This is a well-known dataset with a time-varying “period.”⁷ AR again fails completely. SAR (without a MA component, much less MLE) takes 40 minutes to estimate. AWSOM (in Python) takes less than 9 seconds. SAR misses the marginal distribution (see Fig. 11) but, more importantly, it does not discover any period; that information has to be manually provided, after an initial inspection of the data. Furthermore, SAR can only deal successfully with one period only. AWSOM captures the general periodic trend, with a desirable slight confusion about the “period.”

⁷Signals exhibiting this behavior are often referred to as *cyclical*, as opposed to *periodic*.

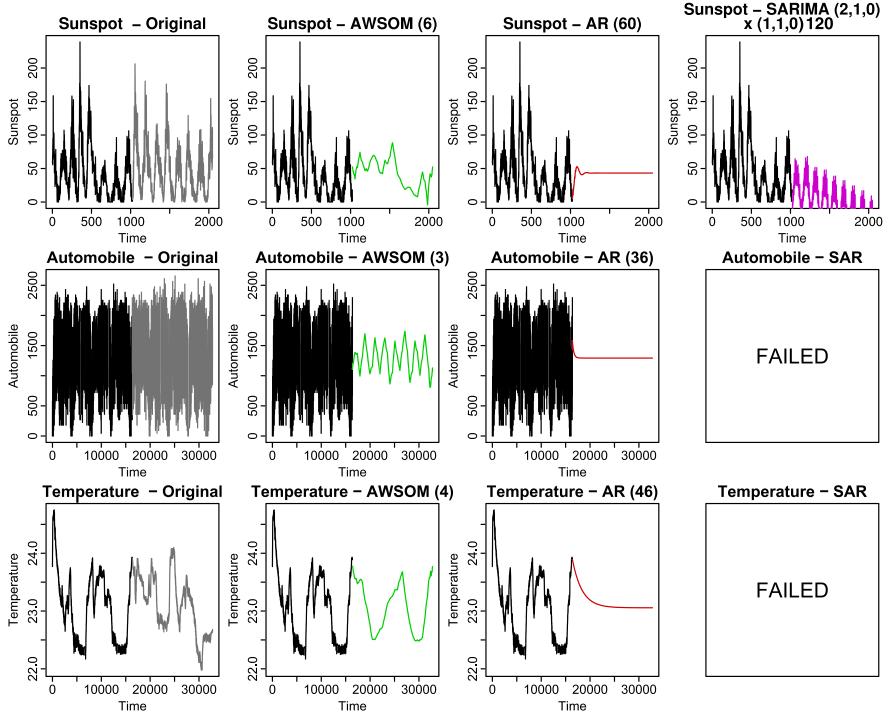


Fig. 8 Forecasts—real datasets. AR fails to detect any trend, while seasonal AR fails to complete or gives a wrong conclusion in $260 \times$ time

First, the log-power diagnostic (Fig. 6(b)) shows some hint of self-similar noise at scales 2–4 (and, indeed, the series has some fluctuation at the month scale). We won't focus on this behavior here (see, however, discussion on Automobile) since the other periodic (or, more accurately, cyclic) components are those that are of the most interest and dominate the series by far.

The wavelet variance (Fig. 9(c)) indeed shows a spike in the vicinity of scale 6 (or window $2^6 = 64$). Each time tick is one month; it is a well-known fact (the so-called “Maunder minimum”) that sunspots cycle at about 9–11.5 years (or, 108–138 months), with an average cycle length of about 10.8 years (approximately 129–130 months). Indeed, we see that the wavelet variance has a peak at windows of 64–128 (scales 6–7), which would correspond to repeating cycles every 128 or more months, as is the case.

Furthermore, the AWSOM coefficients at those scales capture the trend at that granularity (as can be seen immediately from the forecast), with the desirable “confusion” about the period.

Finally, closer examination of the series shows that the peaks at the last third are much lower than the rest of the series. This explains the other peak in Fig. 9(c) at the scales of 9–10, which correspond to a window of about 1/4–1/2 of the total series length.

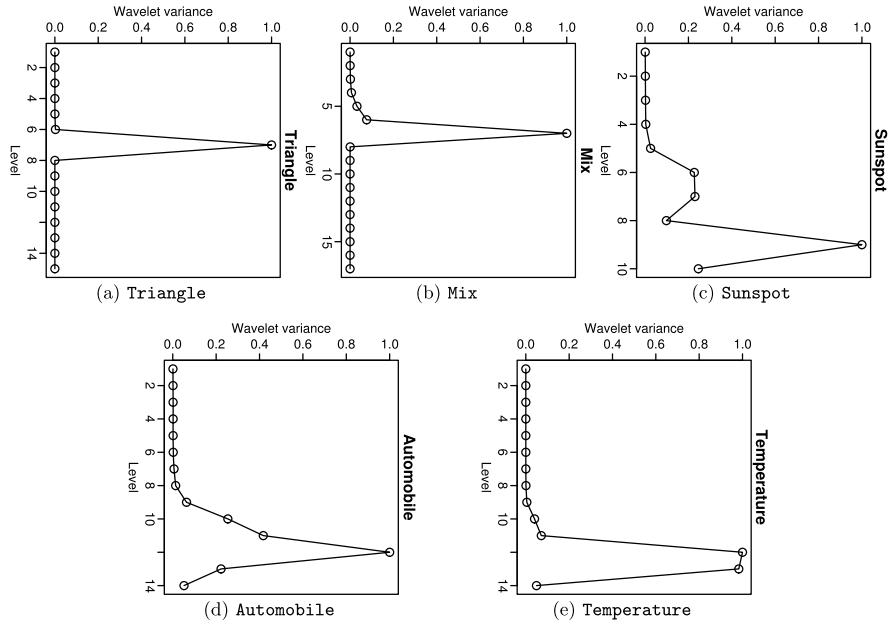


Fig. 9 Wavelet variances (average energy, normalized). The horizontal axis is wavelet level (l , i.e., wavelet window 2^l) and the vertical is wavelet variance (i.e., average of squares of wavelet coefficients for each l). The vertical axis is (linearly) normalized to a maximum of 1. Essentially, values at level l indicate “strength” of the series component at periods in the range $[2^l, 2^{l+1}]$ (see also Appendix B)

Automobile

This dataset has a strongly linear log-power diagnostic in scales 1–6 (Fig. 6(a)). However, the lower frequencies (i.e., larger scales) contain the most energy (see Fig. 9(d)). This indicates we should focus at these scales. The lowest frequency corresponds to a daily periodicity (approximately 4000 points per day, or about 8 periods in the entire series). The next highest frequency corresponds to the morning and afternoon rush-hours (approximately 2000 points per half-day). Also, we would expect to see a rise in the wavelet variance at scales corresponding to windows of $\approx 1000\text{--}2000$ or in the vicinity of scales 10–11. Indeed, this is the case, and at these scales, the AWSOM coefficients capture the periodic components.

Furthermore, there appear to be significant differences in the dips between the two halves (this is clearer in Fig. 1, where some of the noise has been removed), which explains the continued rise up to scale 12 (along with some small frequency leak, as explained in Appendix B). This trend is not repeated frequently enough to be captured by a regression models. However, the variance plot gives us a hint about this and the regression models on the other scales still do their job.

Next, we examine closer the self-similar noise at the hour scale (or high frequencies). In this series, these frequencies (corresponding scales 1–6) can be modeled

Fig. 10

Automobile—generation with fractional noise

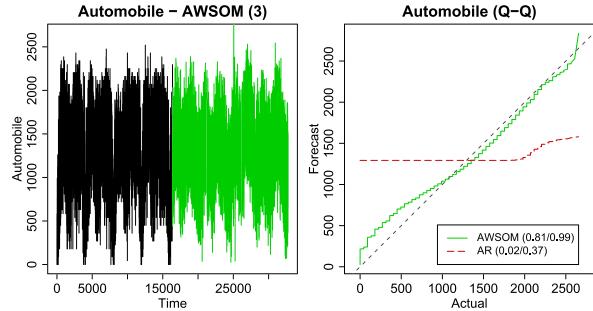
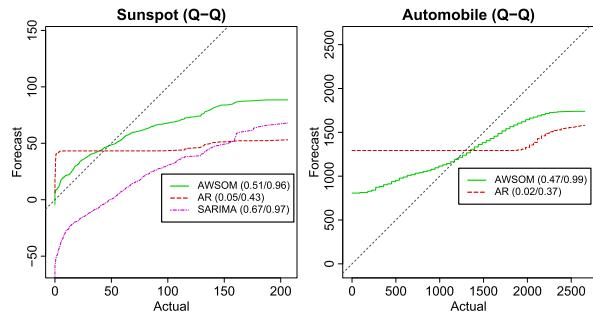


Fig. 11 Marginal Q–Q plots (slope and correlation coefficients in parentheses)



by fractional noise. Figure 10 shows a generated sequence with fractional noise, as identified by AWSOM (compare to Fig. 8, middle row). The fractional difference parameter⁸ is estimated as $\hat{\delta} \equiv -\hat{\alpha}/2 \approx 0.276$ and the amplitude is chosen to match the total variance in those scales.

However, for unsupervised outlier detection, this is not necessary: what would really constitute an outlier would be, for instance, days that (a) do not follow the daily and rush-hour patterns, or (b) whose variance in the fractional noise scales is very different. This can be captured automatically by the series components in the appropriate frequency sub-bands that AWSOM identifies as a periodic component and bursty noise, respectively.

Temperature

This data set consists of temperature measurements (in degrees Celsius) from small sensors that attach to the joystick port.⁹ Each time tick is one second, thus the entire dataset is approximately 10 hours. The interpretation is similar to that of Automobile, except that there isn't a strong noise component at any scale (see Fig. 6(c); the slope is *not* negative, as required for diagnosing self-similar noise). Also, observe that the wavelet variance (Fig. 9(e)) tells us that the strongest trends happen

⁸This parameter is also related to the *Hurst exponent*.

⁹<http://www.ices.cmu.edu/sensornets/>.

at largest scales (from 10 and on, or windows > 1024). In other words, the most interesting activity is at times larger than about half an hour, which is indeed the case.

6 Conclusions

Sensor networks are becoming increasingly popular, thanks to falling prices and increasing storage and processing power. We presented AWSOM, which achieves all of the following goals:

1. *Concise patterns.* AWSOM provides linear models with few coefficients, it can detect arbitrary periodic components, it gives information across several frequencies and it can diagnose self-similarity and long-range dependence.
2. *Streaming framework.* We can update patterns in an “any-time” fashion, with one pass over the data, in time independent of stream size and using $O(\lg N)$ space (where N is the length of the sequence so far). Furthermore, AWSOM can do forecasting (directly, for the estimated model).
3. *Unsupervised operation.* Once we decide the largest AWSOM order, no further intervention is needed; the sensor can be left alone to collect information.

We showed real and synthetic data, where our method captures the periodicities and burstiness, while manually selected AR (or even (S)ARIMA generalizations, which are not suitable for streams with limited resources) fails completely.

AWSOM is an important first step toward hands-off stream mining, combining simplicity with modeling power. Continuous queries are useful for evidence gathering and hypothesis testing *once* we know what we are looking for. AWSOM is the first method to deal directly with the problem of unsupervised stream mining and pattern detection and fill the gap.

Acknowledgements We thank Becky Buchheit for her help with the automobile traffic datasets and Mike Bigrigg for the temperature sensor data.

Appendix A: Auto-Regressive Modeling

In their simplest form, an auto-regressive model of order p , or AR(p) express X_t as a linear combination of previous values, i.e., $X_t = \phi_1 X_{t-1} + \dots + \phi_p X_{t-p} + \epsilon_t$ or, more concisely,

$$\phi(L)X_t = \epsilon_t$$

where L is the lag operator and $\phi(L)$ is a polynomial defined on this operator:

$$LX_t \equiv X_{t-1},$$

$$\phi(L) = 1 - \phi_1 L - \phi_2 L^2 - \dots - \phi_p L^p,$$

and ϵ_t is a white noise process, i.e.,

$$\text{E}[\epsilon_t] = 0 \quad \text{and} \quad \text{Cov}[\epsilon_t, \epsilon_{t-k}] = \begin{cases} \sigma^2 & \text{if } k = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Using least-squares, we can estimate σ^2 from the sum of squared residuals (SSR). This is used as a measure of estimation error; when generating “future” points, ϵ_t is set to $\text{E}[\epsilon_t] \equiv 0$.

The next step up are auto-regressive moving average models. An ARMA(p, q) model expresses values X_t as

$$\phi(L)X_t = \theta(L)\epsilon_t$$

where $\theta(L) = 1 - \theta_1L - \dots - \theta_qL^q$. Estimating the moving average coefficients θ_i is fairly involved. State-of-the-art methods use maximum-likelihood (ML) algorithms, employing iterative methods for nonlinear optimization, whose computational complexity depends exponentially on q .

ARIMA(p, d, q) models are similar to ARMA(p, q) models, but operate on $(1 - L)^d X_t$, i.e., the d th order backward difference of X_t :

$$\phi(L)(1 - L)^d X_t = \theta(L)\epsilon_t.$$

Finally, SARIMA($p, d, q) \times (P, D, Q)T$ models are used to deal with seasonalities, where

$$\phi(L)\Phi(L^T)(1 - L)^d(1 - L^T)^D X_t = \theta(L)\Theta(L^T)\epsilon_t$$

and where the seasonal difference polynomials,

$$\Phi(L^T) = 1 - \Phi_1 L^T - \Phi_2 L^{2T} - \dots - \Phi_P L^{PT},$$

$$\Theta(L^T) = 1 - \Theta_1 L^T - \Theta_2 L^{2T} - \dots - \Theta_Q L^{QT},$$

are similar to $\phi(L)$ and $\theta(L)$ but operate on lags that are multiples of a fixed period T . The value of T is yet another parameter that either needs to be estimated or set based on prior knowledge about the series X_t .

Appendix B: More Wavelet Properties

Frequency Properties

Wavelet filters employed in practice can only approximate an ideal bandpass filter since they are of *finite* length L . The practical implications are that wavelet coefficients at level l correspond roughly to the frequency range $[1/2^{l+1}, 1/2^l]$, or, equivalently, periods in $[2^l, 2^{l+1}]$ (see Fig. 12 for the actual correspondence). This has to be taken into account for precise interpretation of AWSOM models by an expert.

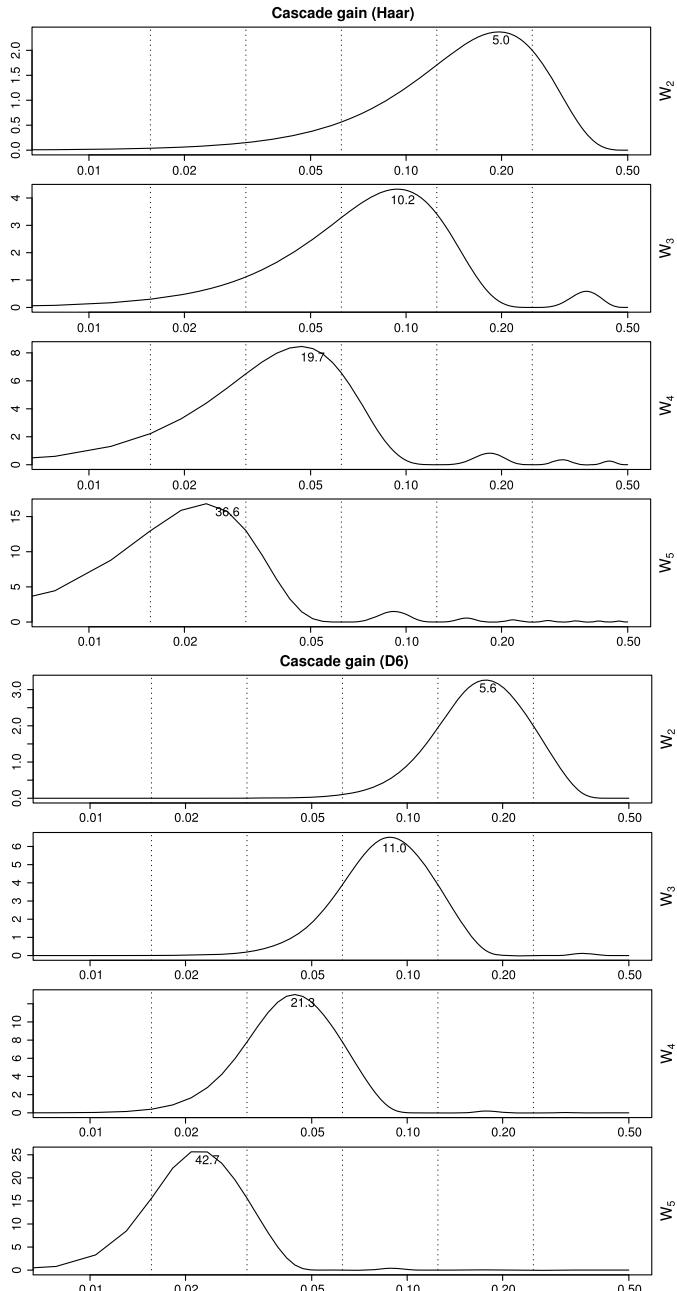


Fig. 12 Illustration of Haar and Daubechies-6 cascade gain (levels 3–5). The *horizontal axis* is frequency and the *curves* show how much of each frequency is “represented” at each wavelet level. As expected, D-6 filters (used in all experiments), have better band-pass properties

Wavelet Variance and Self-Similarity

The wavelet variance decomposes the variance of a sequence across scales. Due to space limitations, we mention basic definitions and facts; details can be found in [29].

Definition 4 (Wavelet Variance) If $\{W_{l,t}\}$ is the DWT of a series $\{X_t\}$ then the wavelet variance \mathcal{V}_l is defined as $\mathcal{V}_l = \text{Var}[W_{l,t}]$.

Under certain general conditions, $\hat{\mathcal{V}}_l = \frac{2^l}{N} \sum_{t=1}^{N/2^l} W_{l,t}^2$ is an *unbiased* estimator of \mathcal{V}_l . Note that the sum is precisely the energy of $\{X_t\}$ at scale l .

Definition 5 (Self-Similar Sequence) A sequence $\{X_t\}$ is said to be self-similar following a pure power-law process if $\mathcal{S}_X(f) \propto |f|^\alpha$, where $-1 < \alpha < 0$ and $\mathcal{S}_X(f)$ is the SDF.¹⁰

It can be shown that $\mathcal{V}_l \approx 2 \int_{1/2^{l+1}}^{1/2^l} \mathcal{S}_X(f) df$, thus if $\{X_t\}$ is self-similar, then

$$\log \mathcal{V}_l \propto l,$$

i.e., the plot of $\log \mathcal{V}_l$ versus the level l should be linear. In fact, the slope of the log-power versus scale plot should be approximately equal to the exponent α . This fact and how to estimate \mathcal{V}_l are what the reader needs to keep in mind.

References

1. M. Akay (ed.), *Time Frequency and Wavelets in Biomedical Signal Processing* (Wiley, New York, 1997)
2. A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom, Characterizing memory requirements for queries over continuous data streams, in *PODS* (2002)
3. B. Babcock, C. Olston, Distributed top- k monitoring, in *Proc. SIGMOD* (2003)
4. J. Beran, *Statistics for Long-Memory Processes* (Chapman & Hall, London, 1994)
5. T. Bollerslev, Generalized autoregressive conditional heteroskedasticity. *J. Econom.* **31**, 307–327 (1986)
6. P. Bonnet, J.E. Gehrke, P. Seshadri, Towards sensor database systems, in *Proc. MDM* (2001)
7. P.J. Brockwell, R.A. Davis, *Time Series: Theory and Methods*, 2nd edn. Springer Series in Statistics (Springer, Berlin, 1991)
8. A. Bulut, A.K. Singh, SWAT: hierarchical stream summarization in large networks, in *Proc. 19th ICDE* (2003)
9. L.R. Carley, G.R. Ganger, D. Nagle, Mems-based integrated-circuit mass-storage systems. *Commun. ACM* **43**(11), 72–80 (2000)

¹⁰The *spectral density function (SDF)* is the Fourier transform of the auto-covariance sequence (ACVS) $S_{X,k} \equiv \text{Cov}[X_t, X_{t-k}]$. Intuitively, it decomposes the variance into frequencies.

10. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S.B. Zdonik, Monitoring streams—a new class of data management applications, in *Proc. VLDB* (2002)
11. Y. Chen, G. Dong, J. Han, B.W. Wah, J. Wang, Multi-dimensional regression analysis of time-series data streams, in *Proc. VLDB* (2002)
12. J. Considine, F. Li, G. Kollios, J.W. Byers, Approximate aggregation techniques for sensor databases, in *Proc. ICDE* (2004)
13. G. Das, K.-I. Lin, H. Mannila, G. Renganathan, P. Smyth, Rule discovery from time series, in *Proc. KDD* (1998)
14. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows, in *Proc. SODA* (2002)
15. M.H. DeGroot, M.J. Schervish, *Probability and Statistics*, 3rd edn. (Addison-Wesley, Reading, 2002)
16. A. Dobro, M.N. Garofalakis, J. Gehrke, R. Rastogi, Processing complex aggregate queries over data streams, in *Proc. SIGMOD* (2002)
17. C. Faloutsos, *Searching Multimedia Databases by Content* (Kluwer Academic, Norwell, 1996)
18. M.N. Garofalakis, P.B. Gibbons, Wavelet synopses with error guarantees, in *Proc. SIGMOD* (2002)
19. J. Gehrke, F. Korn, D. Srivastava, On computing correlated aggregates over continual data streams, in *Proc. SIGMOD* (2001)
20. R. Gencay, F. Selcuk, B. Whitcher, *An Introduction to Wavelets and Other Filtering Methods in Finance and Economics* (Academic Press, San Diego, 2001)
21. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *Proc. VLDB* (2001)
22. S. Guha, N. Koudas, Approximating a data stream for querying and estimation: algorithms and performance evaluation, in *Proc. ICDE* (2002)
23. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, in *Proc. ASPLOS-IX* (2000)
24. P. Indyk, N. Koudas, S. Muthukrishnan, Identifying representative trends in massive time series data sets using sketches, in *Proc. VLDB* (2000)
25. W. Leland, M. Taqqu, W. Willinger, D. Wilson, On the self-similar nature of Ethernet traffic. *IEEE Trans. Netw.* **2**(1), 1–15 (1994)
26. S.R. Madden, M.A. Shah, J.M. Hellerstein, V. Raman, Continuously adaptive continuous queries over streams, in *SIGMOD Conf.* (2002)
27. C. Olston, J. Jiang, J. Widom, Adaptive filters for continuous queries over distributed data streams, in *Proc. SIGMOD* (2003)
28. T. Palpanas, M. Vlachos, E.J. Keogh, D. Gunopoulos, W. Truppel, Online amnesic approximation of streaming time series, in *Proc. ICDE* (2004)
29. D.B. Percival, A.T. Walden, *Wavelet Methods for Time Series Analysis* (Cambridge University Press, Cambridge, 2000)
30. E. Riedel, C. Faloutsos, G.R. Ganger, D. Nagle, Data mining on an OLTP system (nearly) for free, in *SIGMOD Conf.* (2000)
31. Y. Tao, C. Faloutsos, D. Papadias, B. Liu, Prediction and indexing of moving objects with unknown motion patterns, in *Proc. SIGMOD* (2004)
32. A.S. Weigend, N.A. Gerschenfeld, *Time Series Prediction: Forecasting the Future and Understanding the Past* (Addison-Wesley, Reading, 1994)
33. B.-K. Yi, N. Sidiropoulos, T. Johnson, H. Jagadish, C. Faloutsos, A. Biliris, Online data mining for co-evolving time sequences, in *Proc. ICDE* (2000)
34. P. Young, *Recursive Estimation and Time-Series Analysis: An Introduction* (Springer, Berlin, 1984)

35. D. Zhang, D. Gunopulos, V.J. Tsotras, B. Seeger, Temporal aggregation over data streams using multiple granularities, in *Proc. EDBT* (2002)
36. Y. Zhu, D. Shasha, Statstream: statistical monitoring of thousands of data streams in real time, in *Proc. VLDB* (2002)
37. Y. Zhu, D. Shasha, Efficient elastic burst detection in data streams, in *Proc. KDD* (2003)
38. R. Zuidwijk, P. de Zeeuw, Fast algorithm for directional time-scale analysis using wavelets, in *Proc. SPIE, Wavelet Applications in Signal and Image Processing VI*, vol. 3458 (1998)

Conclusions and Looking Forward

Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi

Today, data streaming is a part of the mainstream and several data streaming products are now publicly available. Data streaming algorithms are powering complex event processing, predictive analytics, and big data applications in the cloud. In this final chapter, we provide an overview of current data streaming products, and applications of data streaming to cloud computing, anomaly detection and predictive modeling. We also identify future research directions for mining and doing predictive analytics on data streams, especially in a distributed environment.

1 Data Streaming Products

Given the need for processing data streams in manufacturing, financial trading, logistics, telecom, health monitoring, and web analytics applications, a number of the leading software vendors have launched commercial data streaming products. We describe three prominent products below.

TIBCO Streambase is a high-performance system for rapidly building applications that analyze and act on real-time streaming data. StreamBase's EventFlow language

M. Garofalakis (✉)

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

J. Gehrke

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA
e-mail: johannes@microsoft.com

R. Rastogi

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

represents stream processing flows and operators as graphical elements—operators can be placed on a canvas and connected with arrows that represent the flow of data. Furthermore, StreamBase’s StreamSQL extends the standard SQL querying model and relational operators to also perform processing on continuous data streams. EventFlow and StreamSQL both extend the semantics of standard SQL by adding rich windowing constructs and stream-specific operators. Windows are definable over time or the number of messages, and define the scope of a multi-message operator such as an aggregate or a join. EventFlow and StreamSQL operators provide the capability to filter streams, merge, combine, and correlate multiple streams, and run time-window-based aggregations and computations on real-time streams. In addition, developers can easily extend the set of operators available to either EventFlow or StreamSQL modules by writing their own operators.

IBM System S provides a programming model and an execution platform for user-developed applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous data streams. In System S, users create applications in the form of dataflow graphs consisting of analytics operators interconnected by streams. System S provides a toolkit of type-generic built-in stream processing operators, which include all basic stream-relational operators, as well as a number of plumbing operators (such as stream splitting, demultiplexing, etc.). It also provides an extensible operator framework, which supports the addition of new type-generic and configurable operators.

Microsoft StreamInsight implements a lightweight streaming architecture that supports highly parallel execution of continuous queries over high-speed event data. Developers can write their applications using Microsoft’s .NET language such as Visual C#, leveraging the advanced language platform LINQ (Language Integrated Query) as an embedded query language. By using LINQ, developers can write SQL queries in a declarative fashion that process and correlate data from multiple streams into meaningful results. The optimizer and scheduler of the StreamInsight server in turn ensure optimal query performance. Incoming events are continuously streamed into standing queries in the StreamInsight server, which processes and transforms the data according to the logic defined in each query. The query result at the output can then be used to trigger specific actions. Each event consists of the following parts: (i) An event header that contains metadata defining the event kind (interval vs. point events) and one or more timestamps that define the time interval for the event, and (ii) The payload of an event is a .NET data structure that contains the data associated with the event. StreamInsight provides the following query functionality: (i) Filter operation to express Boolean predicates over the event payload and discard events that do not satisfy the predicates, (ii) Grouping operation that partitions the incoming stream based on event properties such as location or ID and then applies other operations or complete query fragments to each group separately, (iii) Hopping and sliding windows to define windows over event streams—a sliding window contains events within the last X time units, at each point in time, (iv) Built-in aggregations for sum, count, min, max, and average that typically operate on time windows, (v) TopK operation to identify heavy hitters in an event stream, (vi) A powerful join

operation that matches events from two sources if their times overlap and executes the join predicate specified on the payload fields, (vii) A union operation that multiplexes several input streams into a single output stream, and (viii) User-defined functions.

2 Data Streaming in the Cloud

Many internet applications such as web site statistics monitoring and analytics, intrusion detection systems and spam filters have real time processing requirements. To provide real-time analytics functionality to such applications, a number of systems for processing data streams in the cloud have been developed recently. We describe a few of these below.

Spark Streaming extends Apache Spark for doing large scale stream processing. It chops up the live stream into batches of X seconds. Each batch of data is treated as a Resilient Distributed Dataset (RDD) and processed using RDD operations such as map, count, join, etc. Finally, the processed results of the RDD operations are returned in batches and can be persisted in HDFS. Thus, Spark represents a stream of data as a sequence of RDDs (referred to as DStream) and applies transformations that modify data from one DStream to another using standard RDD operations. Spark Streaming also has support for defining Windowed DStreams that gather together data over a sliding window. Each window has two parameters, a window length and a sliding interval. At each time t, the function for time t is applied to the union of all RDDs of the parent DStream between times t and (t—window length). One can use Spark Streaming to get hashtags from a Twitter stream, count the number of hashtags in each window, or join incoming tweets with a file containing spam keywords to filter out bad tweets.

Amazon Kinesis enables users to collect and process large streams of data records in real time. A producer puts data records into Amazon Kinesis streams. For example, a web server sending log data to an Amazon Kinesis stream is a producer. Each data record consists of a sequence number, a partition key, and a data blob that is not interpreted by Kinesis. The data records in a stream are distributed into multiple shards, using the partition key associated with each data record to determine which shard a given data record belongs to. Specifically, an MD5 hash function is used to map partition keys to 128-bit integer values and to map associated data records to shards. When a stream is created, the number of shards for the stream are specified by the application. Each consumer reads data records from a particular shard.

Apache Kafka is a high-throughput publish-subscribe messaging system. Kafka maintains feeds of messages in categories called topics. Producers publish messages to a topic that are delivered to consumers who have subscribed to the topic. Each topic consists of multiple partitions and producers can specify which partition each message within a topic is assigned to. Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer

instance within each subscribing consumer group. Kafka provides both ordering guarantees and load balancing over a pool of consumer processes by assigning partitions in the topic to consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. Kafka provides a total order over messages only within a partition, not between different partitions in a topic. Use cases for Apache Kafka include messaging, website activity tracking (page views, searches), and stream processing.

Apache Storm makes it easy to reliably process unbounded streams of data. The basic primitives Storm provides for doing stream transformations are “spouts” and “bolts”. A spout is a source of streams. For example, a spout may connect to the Twitter API and emit a stream of tweets. A bolt consumes any number of input streams, does some processing, and possibly emits new streams. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, etc. For example, a bolt may perform complex stream transformations like computing a stream of trending topics from a stream of tweets. Networks of spouts and bolts are packaged into a “topology” to do real-time computation. A topology is a graph of stream transformations where each node is a spout or bolt. Links between nodes in the topology indicate how tuples should be passed around between nodes. For example, when a spout or bolt emits a tuple, it sends the tuple to every bolt to which it has an outgoing edge. Thus, a storm topology consumes streams of data and processes those streams in arbitrary complex ways, repartitioning the streams between each stage of the computation as needed.

3 Complex Event Processing

Complex event processing (CEP) techniques discover complex events by analyzing patterns and correlating other events. For instance, a CEP application may analyze tweet events in a Twitter stream to detect complex events such as an earthquake or a plane accident. Another CEP application in the manufacturing domain may generate an alert when a measurement exceeds a predefined threshold of time, temperature, or other value. CEP applications include algorithmic stock-trading, network anomaly detection, credit-card fraud detection, demand spike detection in retail, and security monitoring.

CEP applications may require complex pattern matching over high-speed data streams. For instance, Network intrusion detection systems (NIDS) like Snort perform deep packet inspection on an IP traffic stream to check if the packet payload (and header) matches a given set of signatures of well known security threats (e.g., viruses, worms). The signature patterns are frequently specified using general regular expressions because it is a more expressive pattern language compared to strings. Thus, matching thousands of regular expressions on data streams is a critical capability in network security applications. Current matching algorithms employ finite automata-based representations to detect regular expression patterns.

Another example in the retail space is detection of spikes in product demand. Such spikes can occur because of different types of events, e.g., a snowstorm can trigger the sale of snow shovels. Similarly, there may be a jump in the sale of textbooks when students go back to school. One strategy to detect sudden demand spikes is to maintain a sliding window and compute the mean demand and standard deviation over the window. If the current observed demand deviates from the mean by more than two standard deviations then we can generate a demand spike event.

Numenta's Grok application employs an innovative approach to detect anomalies in numeric streaming data. It continuously trains a machine learning model on a sliding window of the most recently observed data and uses the model to predict the next most likely value based on the sliding window of previous values. Specifically, the model returns the probability that the next value in the data stream is the observed value. If multiple contiguous stream values have low occurrence probabilities then Grok generates an anomaly event. Grok is being used to detect anomalies in streaming metrics (e.g., CPU utilization, disk writes) from server clusters.

4 Big Data and Predictive Modeling

As businesses increasingly computerize and automate their operations, they have the ability to collect massive amounts of data and leverage the data to automate decision making at every step. Consequently, Big Data analytics and Predictive modeling is ubiquitous today across a wide range of domains such as telecom, healthcare, internet, finance, retail, media, transportation and manufacturing, and is leading to: (i) Improved customer experience—Online content providers and web search companies analyze historical customer actions (e.g., clicks, searches, browsed pages) to learn about customer preferences and then show relevant content and search results that match a user's interests, (ii) Reduced costs—in transportation and manufacturing, analysis of past equipment operations data enables proactive prediction of failures and servicing of equipment prior to failures, and (iii) Higher revenues—in online advertising and e-commerce, targeting customers with relevant ads and recommending relevant products can lead to higher ad clicks and product purchases.

Data streaming algorithms are critical for Big Data analytics and Predictive modeling. Below, we list applications of data streaming algorithms in various Predictive modeling steps:

- *Data cleaning.* Collected data is frequently dirty with errors, outliers and missing values. For a numeric attribute, values greater than a certain threshold (e.g., 98th percentile, mean plus two times standard deviation) or smaller than a certain threshold (e.g., 2nd percentile, mean minus two times standard deviation) can be considered as outliers. Similarly, for a categorical attribute, values that occur very frequently or infrequently can be regarded as outliers. Single-pass algorithms for computing a wide variety of data statistics such as percentiles, mean and standard deviation for numeric attributes, and value frequencies for categorical attributes can help to detect outliers and prune them, thus ensuring high data quality.

- *Data statistics and visualization.* Univariate data statistics provide insights into attribute and target data distributions—these include different percentiles, mean and standard deviation for numeric attributes, and cardinality and top frequent values/keywords for categorical and text attributes. Text attributes, in particular, may contain millions of keywords making it difficult to store counts for individual keywords in main memory. These analyses require scalable algorithms for constructing histograms over numeric attribute values, and computing attribute value frequencies and cardinality. Bivariate data statistics shed light on the degree of correlation between attribute and target values. Commonly used correlation metrics include Pearson’s Correlation Coefficient for numeric attributes, and information gain or mutual information for categorical and text attributes. While Pearson’s Coefficient can be computed efficiently in one pass over the data, information gain and mutual information estimation requires calculating attribute-target value pair frequencies using streaming techniques.
- *Feature engineering.* To obtain models with high predictive accuracy, raw data needs to be transformed into higher level representations with predictive power. For instance, individual pixels within an image may not have much signal but higher level aggregations such as color histograms or shapes in the image may be a lot more predictive. A common transformation is numeric attribute binning which can be carried out on large data sets using stream quantile computation algorithms. Another option is to construct interaction features over attribute pairs, e.g., in online advertising, an interaction feature involving user gender and ad category has more signal compared to the individual attributes themselves (since women are more likely to click on jewelry or beauty product ads while men are more likely to click on ads related to automotive parts). Here, min-wise hashing techniques can be used to efficiently compute counts for frequent attribute value pairs belonging to interaction features. More recently, deep learning has shown promise for unsupervised learning of higher level features in speech recognition, computer vision and natural language processing applications. Efficient algorithms for learning deep neural networks on large datasets is an active area of research.
- *Feature selection.* Noisy and redundant features can cause trained models to overfit the data, thus adversely impacting their accuracy. In order to prune such features, we need scalable algorithms for (i) computing correlations between attributes and the target using metrics such as Pearson’s coefficient, information gain or mutual information, and selecting highly correlated features with predictive power, (ii) computing pairwise attribute correlations and dropping redundant features that are highly correlated with other features, and (iii) reducing data dimensionality using techniques such as PCA, SVD and matrix factorization.
- *Model training.* Online learning algorithms such as stochastic gradient descent (SGD) can be used to train linear and logistic regression models, and perform tasks such as matrix factorization. SGD updates model parameters with the gradient computed for each example as opposed to the entire dataset as is done by traditional batch learning algorithms like BFGS. For clustering problems, proposed data streaming algorithms first independently cluster data partitions in an

initial pass and subsequently cluster the centroids for each partition in a second pass. Note that a possible strategy for scaling learning algorithms is to train models on a random sample of the data. Random samples can be obtained in a single pass over the data using reservoir sampling. However, the random sampling approach does not consider the entire data and so could hurt model accuracy. Scalable training and inference algorithms for complex ML models such as decision trees, random forests, neural networks and graphical models is an active area of research.

In addition to data streaming algorithms, scaling predictive modeling to large terabyte datasets also requires parallelizing the algorithms over large machine clusters. There are two main parallelization paradigms proposed in the literature.

- *Loosely-coupled paradigm.* In this paradigm, data is partitioned across machines, the algorithm is run locally on each machine and then the data summaries/synopses structures from the different machines are combined into a single summary/synopsis structure. Thus, the loosely-coupled paradigm fits well with Map-Reduce style computation with reducers combining the synopses computed by the mappers. Note that multiple Map-Reduce iterations may be performed in this paradigm with the synopsis computed at the end of each iteration used to initialize the synopsis at the beginning of the next iteration.

As an example, consider k -means clustering. The synopsis contains the k cluster centroids. Each mapper assigns points in its data partition to the closest centroid and computes k new centroids for the new assignment. The mappers transmit the new cluster centroids along with the size of each cluster to the reducers that combine the centroids to compute k new centroids using weighted averaging.

The loosely-coupled paradigm is especially well-suited for parallelizing algorithms where synopses satisfy the *composability* property, that is, sketches computed on individual data partitions can be combined to yield a single synopsis for the entire dataset. Several synopses structures such as the Flajolet–Martin (FM) sketch and the Count-Min sketch satisfy the composability property. Algorithms that maintain such synopses can be naturally parallelized using Map-Reduce with mappers computing synopses for each data partition and the reducer composing a single synopsis from the synopses computed by mappers.

- *Tightly-coupled paradigm.* In many instances, synopses may not satisfy the composability property, thus making the loosely-coupled paradigm inapplicable for parallelization. For example, in online machine learning algorithms like SGD, parameter values computed for each data partition cannot be easily combined to yield the same parameter values had the algorithm been run on the full dataset. To handle such scenarios, tight coupling between the parallel instances is needed. In the tightly-coupled paradigm, the synopsis structure is replicated across multiple machines, and the synopsis replicas are kept synchronized using distributed protocols while the algorithm is running on each machine.

A popular realization of the tightly-coupled paradigm for ML algorithms is through a centralized parameter server. The parameter server synchronizes the values of parameter replicas distributed across the machines using asynchronous

messages. Specifically, each time a parameter value is updated on a machine, the change in the parameter value is propagated to all the replicas through the parameter server. Furthermore, in order to reduce communication, changes are only propagated if they exceed a certain threshold value.

Scaling ML algorithms through distributed implementations on machine clusters is currently an active area of research.

Popular systems for statistical and exploratory data analysis such as R load the entire dataset into main memory. Consequently, these systems are incapable of analyzing large terabyte datasets that do not fit in memory. To fill the gap, a number of systems for doing predictive analytics on big data have been developed in recent years. These systems rely on online algorithms and parallelization to different degrees in order to handle large datasets. We describe the salient characteristics of some prominent systems below.

Vowpal Wabbit (VW) supports training of a wide spectrum of ML models: linear models with a variety of loss functions like squared, logistic, hinge and quantile loss, matrix factorization models and latent Dirichlet allocation (LDA) models. To scale to terabyte size datasets, it uses the online SGD algorithm to learn model parameters. Furthermore, it employs a parallel implementation of SGD based on the loosely-coupled paradigm that combines parameter values computed on each data partition using simple averaging.

GraphLab provides a high level programming interface for rapid development of distributed ML algorithms based on the tightly-coupled paradigm. Users specify dependencies among ML model parameters using a graph abstraction, the logic for updating parameter values based on those of neighbors and rules for propagating parameter updates to neighbors in the graph. The GraphLab distributed infrastructure partitions parameters across machines to minimize communication, appropriately updates parameter values and transparently propagates parameter updates. GraphLab has a large collection of ML methods already implemented such clustering, matrix factorization and LDA.

Skytree includes many ML methods such as k -means clustering, linear regression, Gradient Boosted trees and Random forests. Through a combination of new efficient ML algorithms and parallelization, Skytree is able to achieve significant speedups on massive datasets.

Azure ML supports a broad range of ML modeling methods including scalable Boosted Decision trees, deep neural networks, logistic and linear regression and clustering using a variant of the standard k -means approach. The linear regression implementation is based on the online SGD algorithm while the logistic regression implementation relies on the batch BFGS learning algorithm.

Apache Mahout offers a scalable machine learning library with support for k -means clustering, matrix factorization, LDA, logistic regression and random forests. Mahout implements a streaming k -means algorithm that processes data points one-by-one and makes only one pass through the data. The algorithm maintains a set of cluster centroids in memory along with a distance threshold parameter.

For a new data point, if its distance from the closest cluster centroid is less than the threshold, then the point is simply assigned to the closest cluster (and its centroid is updated). However, if the distance to the closest centroid is greater than the threshold, then the point starts a new cluster. Finally, if the number of clusters exceeds the memory size, then the distance threshold is increased and existing clusters are merged. The Mahout implementation of logistic regression and matrix factorization uses the online SGD algorithm.

Spark MLlib is Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression clustering, collaborative filtering, and dimensionality reduction. MLlib computes basic statistics such as mean and variance for numeric attributes, and correlations based on Pearson's Coefficient and Pearson's chi-squared tests. For classification and regression, it supports linear models with a variety of loss functions such as squared, logistic and hinge with L1 and L2 regularization, decision trees, and naïve Bayes. MLlib supports SGD and L-BFGS optimization methods to compute parameter values for linear models that minimize the loss function value. MLlib also supports collaborative filtering using the alternating least squares (ALS) algorithm to learn latent factors, k -means clustering, and dimensionality reduction methods such as SVD and PCA.