View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Bilkent University Institutional Repository

Discriminative Fine-Grained Mixing for Adaptive Compression of Data Streams

Buğra Gedik, *Member, IEEE*

Abstract—This paper introduces an adaptive compression algorithm for transfer of data streams across operators in stream processing systems. The algorithm is adaptive in the sense that it can adjust the amount of compression applied based on the bandwidth, Cpu, and workload availability. It is discriminative in the sense that it can judiciously apply partial compression by selecting a subset of attributes that can provide good reduction in the used bandwidth at a low cost. The algorithm relies on the significant differences that exist among stream attributes with respect to their relative sizes, compression ratios, compression costs, and their amenability to application of custom compressors. As part of this study, we present a modeling of uniform and discriminative mixing, and provide various greedy algorithms and associated metrics to locate an effective setting when model parameters are available at run-time. Furthermore, we provide online and adaptive algorithms for real-world systems in which system parameters that can be measured at run-time are limited. We present a detailed experimental study that illustrates the superiority of discriminative mixing over uniform mixing.

Index Terms—stream compression; adaptive compression

1 Introduction

In today's highly instrumented and interconnected world, there is a deluge of data coming from various software and hardware sensors. This data is often in the form of continuous streams. Examples can be found in several domains, such as financial markets, telecommunications, surveillance, manufacturing, and healthcare. Accordingly, there is an increasing need to gather and analyze data streams in near real-time to extract insights and detect emerging patterns and outliers. Stream processing systems [6], [1], [26], [11], [32], [29] enable carrying out these tasks in an efficient and scalable manner, by taking data streams through a network of operators placed on a set of distributed hosts.

In the context of a stream processing system, a data stream is defined as a potentially infinite series of time ordered tuples. Typically, a stream has a well defined schema, which consists of a list of typed attributes defined at application development time [14]. Stream connections among operators that are placed on different hosts is a common occurrence in stream processing systems. Furthermore, the rate of such inter-operator streams is usually very high close to the ingestion point, since most streaming applications perform *progressive filtering* [28]. Such filtering involves using computationally cheap analytics close to the ingestion point and progressively increasing the complexity as the data rates reduce towards the end of the operator data flow graph.

In this work we investigate the problem of adaptive data stream compression, which is a critical functional need in data stream processing systems. As we have outlined, close to the data ingestion point both the computational capacity and the network bandwidth are scarce resources. As such, reducing the rate of data streams by applying compression, without making the Cpu a bottleneck, is a critical capability in increasing the throughput of streaming applications.

• B. Gedik is at Bilkent Univ., Turkey. E-mail: bgedik@cs.bilkent.edu.

Motivated by this need, we develop an adaptive data stream compression scheme called *discriminative fine-grained mixing* (*DFGM*). In its essence, DFGM applies compression judiciously, by determining the best subset of tuple attributes to compress, the best compression algorithms to use, and the right *mixing ratio* to apply. It aims to best utilize the bandwidth and Cpu utilization, with the ultimate goal of maximizing throughput. DFGM takes advantage of the significantly different characteristics of the stream attributes, with respect to compression rate, compression cost, relative size, and suitability of different compression algorithms. Furthermore, through its adaptive nature, it adjusts the level of compression performed based on the changes in the bandwidth, Cpu, and workload availability.

Our work is highly influenced by the fine-grained mixing (FGM) approach of Pu and Singaravelu [20], as well as compression in column oriented databases [3]. FGM [20] is designed for general purpose data transfers, where no assumptions are made about the contents of the data streams. The main idea is to arbitrate between compression and no compression at a very low level, resulting in partial compression of the stream when there is not enough Cpu to perform full compression. The *mixing ratio* can be defined as the average fraction of data blocks that are compressed, even though such a parameter is not explicitly studied in [20].

Since data streams in stream processing systems contain a list of typed attributes, in this work we take advantage of this structure to develop a discriminative fine-grained mixing approach. As shown in the context of column-oriented databases [3], within a single column (attribute), there is often significant repetition. Furthermore, certain kinds of stream attributes (e.g., sequence numbers, Boolean and Enum types, etc.) can be compressed very cheaply with custom compressors. In this work, we take advantage of these properties to provide an adaptive compression scheme based on discriminative mixing, which outperforms uniform mixing.

Digital Object Indentifier 10.1109/TC.2013.103 0018-9340/13/$31.00 © 2013 IEEE

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

In particular, we make the following contributions:

• We provide a modeling of fine-grained mixing and give a formula for the optimal mixing ratio.

• We extend our model to discriminative mixing and formalize an optimization problem.

• We develop several heuristic methods for finding an effective configuration for discriminative mixing, as the brute-force approach is too expensive for streams with many attributes. Our heuristic methods assume that all model parameters can be measured at run-time.

• We develop an online algorithm as well as an online and adaptive algorithm for systems that do not have explicit access to all model parameters. These algorithms make increasing sacrifices in terms of solution optimality, but are more suitable for real-world deployments in stream processing systems.

• We provide an evaluation of our techniques that showcase their effectiveness in terms of throughput as well as bandwidth and Cpu utilization. We use both model-based experiments as well as an implementation that runs on real-world streaming data.

The rest of the paper is organized as follows. Section 2 gives the preliminaries on FGM, including the optimal mixing ratio. Section 3 introduces DFGM and provides several heuristic model-based algorithms, as well as online and adaptive algorithms. Section 4 gives details about our implementation of the DFGM algorithm. Experimental results are presened in Section 5. Section 6 gives the related work. Section 7 discusses future work and Section 8 concludes the paper.

2 Preliminaries

We start by introducing the basic notation. We then identify when FGM can be superior to switching between two modes of all-compress and no-compress. Finally, we provide a formula for the optimal mixing ratio.

2.1 Basic Notation

We denote by T the *throughput* in terms of bytes/s. We denote by p the *mixing ratio* ($0 \le p \le 1$). Mixing ratio represents the ratio of the number of compressed tuples to the total number of tuples. We use r to represent the compression ratio, where $0 < r \le 1$. The compression ratio is the ratio of the size of the compressed data to the size of the original data.

We use c to denote different kinds of computation costs. Concretely, we have:

- *Compression cost*, c_c: cost of compressing tuples.
- *Submission cost*, c_s: cost of submitting tuples.
- *Application cost*, c_p: cost of application related work.

All costs are per-byte. The application cost covers the work done on tuples before they are submitted for transmission. Submission includes the cost of taking tuples through the submission process (the transport stack).

We denote by C the total available computation capacity per second ($0 \le C \le 1$). All computation costs, that is c_c, c_s, and c_p, are also in the range $[0, 1]$. Finally, we denote by B the available bandwidth in terms of bytes/s.

2.2 Fine-grained Mixing

The bandwidth and processing constraints must be satisfied by FGM. Concretely, we have:

$$\begin{aligned} c(p) &\le C/T \text{ (processing constraint), and} \\ b(p) &\le B/T \text{ (bandwidth constraint),} \end{aligned}$$

where $c(p)$ is the per-byte processing cost for a given value of the mixing ratio and $b(p)$ is the per-byte bandwidth cost for the same. We have:

$$\begin{aligned} c(p) &= p \cdot (c_p + c_c + r \cdot c_s) + (1-p) \cdot (c_p + c_s) \\ b(p) &= p \cdot r + (1-p) \end{aligned}$$

The per-byte processing cost simply includes the per-byte processing cost for uncompressed tuples ($c_p + c_s$, since it only involves processing and submission) plus the cost for compressed tuples ($c_p + c_c + r \cdot c_s$, since it involves processing, compression, and submission). The former is scaled with $1-p$ as that is the ratio of the uncompressed tuples, and the latter is scaled with p. Note that per-byte processing cost of compressed tuples have $r \cdot c_s$ as the submission cost, since compression reduces the amount of data to be submitted.

The per-byte bandwidth cost includes the per-byte bandwidth cost of sending an uncompressed tuple (simply 1) plus the cost for compressed tuples (simply r). The former is scaled with $1-p$ as that is the ratio of the uncompressed tuples, and the latter is scaled with p.

With these definitions at hand, the throughput that can be achieved for a given value p of the mixing ratio is denoted by $T(p)$, and is defined as follows:

$$T(p) = \min\left(\frac{C}{c(p)}, \frac{B}{b(p)}\right) \tag{1}$$

Assuming workload availability, Equation 1 follows, as either the computation or the bandwidth becomes a bottleneck, and the throughput is limited by whichever becomes the bottleneck. Note that increasing p means we are compressing more tuples and as such the computational cost increases. We have two special cases: $T_C = T(1)$, throughput for all-compress; and $T_N = T(0)$, throughput for no-compress.

As a special case of Equation 1, we have:

$$\begin{aligned} T_C &= \min\left(\frac{C}{c_p + c_c + r \cdot c_s}, \frac{B}{r}\right) \\ T_N &= \min\left(\frac{C}{c_p + c_s}, B\right) \end{aligned}$$

2.3 Benefit Analysis

An important topic is to determine when FGM brings additional benefits in terms of the throughput. For this purpose, we define few Boolean variables:

- K_C^{cpu}: computation is bottleneck for all-compress
- K_N^{cpu}: computation is bottleneck for no-compress
- K_C^{bwh}: bandwidth is bottleneck for all-compress
- K_N^{bwh}: bandwidth is bottleneck for no-compress

Again, we are assuming that there is sufficient workload to saturate either Cpu or bandwidth. We have:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

$$K_C^{\text{cpu}} \equiv U_C^{\text{bwh}} < 1 \text{ and } K_N^{\text{bwh}} \equiv U_N^{\text{cpu}} < 1 \quad (2)$$

In Equation 2, U_C^{bwh} represents the bandwidth utilization for all-compress, assuming infinite workload availability. The computation is the bottleneck for all-compress if and only if the bandwidth utilization is below 1. In Equation 2, U_N^{cpu} represents the Cpu capacity utilized for no-compress, assuming infinite workload availability. The bandwidth is the bottleneck for no-compress if and only if the Cpu utilization is below 1. We have:

$$K_C^{\text{bwh}} \equiv \neg\, K_C^{\text{cpu}} \text{ and } K_N^{\text{cpu}} \equiv \neg\, K_N^{\text{bwh}} \quad (3)$$

$$K_N^{\text{cpu}} \rightarrow K_C^{\text{cpu}} \text{ and } K_C^{\text{bhw}} \rightarrow K_N^{\text{bhw}} \quad (4)$$

Equation 3 follows, as the system can have a single bottleneck at a time. Equation 4 follows from a simple observation: If the computation is the bottleneck for no-compress, then it must be a bottleneck for all-compress as well, since compression increases the computation cost (we assume[1] that $c_c/c_s > 1 - r$)).

The utilizations are defined as follows:

$$U_N^{\text{cpu}} = \min\left(1, \frac{(c_p + c_s) \cdot B}{C}\right) \quad (5)$$

$$U_C^{\text{bwh}} = \min\left(1, \frac{r \cdot C}{(c_p + c_c + r \cdot c_s) \cdot B}\right) \quad (6)$$

In Equation 5, we assume no-compress and there are two cases. If the bandwidth is the bottleneck, then the throughput is given by B and thus the computational cost is $B \cdot (c_p + c_s)$, leading to a utilization value of $\frac{B \cdot (c_p + c_s)}{C}$. If the computation is the bottleneck, then Cpu utilization equals to 1.

In Equation 6, we assume all-compress and there are two cases as well. If the computation is the bottleneck, then the throughput is given by $C/(c_p + c_c + r \cdot c_s)$ and the bandwidth cost is r times the throughput, leading to a utilization value of $\frac{r \cdot C/(c_p + c_c + r \cdot c_s)}{B}$. If the bandwidth is the bottleneck, then bandwidth utilization equals to 1.

Let T^* denote the optimal throughput that can be achieved with FGM. Table 1 shows all possible scenarios and lists the conditions under which all-compress or no-compress approaches can attain optimality. It also shows the scenarios under which FGM can provide an advantage over switching between no-compress and all-compress. Such throughput advantage has been shown empirically [20].

Table 1 shows all possible scenarios.The first row of the table represents the case when computation is not the bottleneck for the all-compress scenario but the bandwidth is the bottleneck for the no-compress scenario. In this case, the all-compress approach achieves optimal throughput. The second row of the table represents the case when computation is the bottleneck for the all-compress scenario but the bandwidth is not the bottleneck for the no-compress scenario. In this case, the no-compress approach achieves optimal throughput.

1. $c_c > c_s$, that is the cost of compression is larger than the cost of submission, is sufficient to satisfy this, which is typical.

K_C^{cpu}	K_N^{cpu}	K_C^{bwh}	K_N^{bwh}	$T^* = T_C$	$T^* = T_N$
×	×	✓	✓	✓	×
All-compress is the optimal choice if Cpu is not the bottleneck, but the bandwidth is.					
✓	✓	×	×	×	✓
No-compress is the optimal choice if Cpu is the bottleneck, but the bandwidth is not.					
✓	×	×	✓	×	×
Neither all-compress nor no-compress is optimal if Cpu is bottleneck for all-compress and bandwidth for no-compress.					
×	×	×	×	✓	✓
Both all-compress and no-compress are optimal if workload is the limiting factor (no bottlenecks)					

TABLE 1: Optimality of no-compress and all-compress under different scenarios (same color columns are for dual variables.)

The most interesting case is represented by the third row, which happens when *computation is the bottleneck for the all-compress scenario and the bandwidth is the bottleneck for the no-compress scenario*. In this case, neither the all-compress nor the no-compress can achieve optimal throughput. This is where FGM can provide superior performance compared to approaches that switch between no-compress and all-compress.

Finally, the last row of the table shows the case when there are no Cpu or bandwidth bottlenecks. This happens when the workload availability is the bottleneck. In this case, both no-compress and all-compress are optimal, but they make different trade-offs in terms of the load imposed on the Cpu and the bandwidth. For instance, no-compress will achieve optimal throughput using more bandwidth, whereas all-compress will achieve optimal throughput using more Cpu.

2.4 Optimal mixing ratio

We find the mixing ratio that achieves the optimal throughput based on the following theorem.

Theorem 1: The mixing ratio, p^*, maximizing the throughput of FGM for $K_C^{\text{cpu}} \wedge K_N^{\text{bwd}}$ is given by:

$$p^* = \frac{1}{1 + r \cdot \frac{(1 - U_C^{\text{bwh}})}{U_C^{\text{bwh}} \cdot (1 - U_N^{\text{cpu}})}} \quad (7)$$

Proof: Let $U_N^{\text{cpu}}(p)$ be the computation capacity utilized by the non-compressed portion of FGM for a given value of the mixing ratio p. Similarly, let $U_C^{\text{cpu}}(p)$ be the computation capacity utilized by the compressed portion of FGM for a given value of the mixing ratio p. We use $U_{NC}^{\text{cpu}}(p) = U_N^{\text{cpu}}(p) + U_C^{\text{cpu}}(p)$ to denote the total computation capacity utilized by FGM. We use similar notation for throughputs $T_{NC}(p)$ (total throughput), $T_N(p)$ (throughput due to non-compressed data), and $T_C(p)$ (throughput due to compressed data).

Assume no-compress approach for the last row from Table 1 as the baseline for FGM, that is $p = 0$. We have $U_N^{\text{bwh}}(0) = 1$, $U_N^{\text{cpu}}(0) = U_N^{\text{cpu}} < 1$, and $T_N(0) = B$. From this state, we can set $p = p^*$ such that this corresponds to taking ϵ from the bandwidth utilization $U_N^{\text{bwh}}(0)$ of the $p = 0$ state and giving it to the bandwidth utilization $U_C^{\text{bwh}}(p^*)$ of the $p = p^*$ state. Thus we have $U_N^{\text{bwh}}(p^*) = 1 - \epsilon$ and $U_C^{\text{bwh}}(p^*) = \epsilon$. For optimality of throughput, ϵ should be made as large as possible, as long as there is enough computational capacity. Initially

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

$U^{\text{cpu}}_{NC}(0) = U^{\text{cpu}}_{N}(0) < 1$, and thus we need to have $U^{\text{cpu}}_{NC}(p^*) = 1$ to maximize the throughput.

Moving ϵ unit of bandwidth utilization from no compression to compression will increase the throughput to $T_{NC}(p^*) = B \cdot (1-\epsilon) + B \cdot \epsilon / r = B \cdot (1 + \epsilon \cdot (1/r - 1))$, since compression uses bandwidth more efficiently. The bandwidth utilization is still kept at its maximum. Moreover, the computation utilization of the non-compressed part is reduced to $U^{\text{cpu}}_{N}(p^*) = (1-\epsilon) \cdot U^{\text{cpu}}_{N}$. On the other hand, the computation utilization of the compressed part is increased to $U^{\text{cpu}}_{C}(p^*) = \epsilon / U^{\text{bwh}}_{C}$. The former follows as the computation cost is linear to the bandwidth for the case of no compression. The latter follows, as to use ϵ amount of bandwidth utilization with the compressed approach, one needs to achieve $\epsilon \cdot B/r$ throughput, which means $(\epsilon \cdot B/r) \cdot (c_s + c_c + r \cdot c_s)$ computation capacity and thus $\epsilon / (\frac{r \cdot C}{B \cdot (c_s + c_c + r \cdot c_s)}) = \epsilon / U^{\text{bwh}}_{C}$ computation utilization.

The sum of the computation utilizations for the no compression and compression parts should be 1 for $p = p^*$, so as to use all the available resources to maximize the throughout. Thus, $U^{\text{cpu}}_{NC}(p^*) = U^{\text{cpu}}_{N}(p^*) + U^{\text{cpu}}_{C}(p^*) = 1$. This means $(1-\epsilon) \cdot U^{\text{cpu}}_{N} + \epsilon / U^{\text{bwh}}_{C} = 1$. Solving this, we get $\epsilon = \frac{(1 - U^{\text{cpu}}_{N}) \cdot U^{\text{bwh}}_{C}}{1 - U^{\text{cpu}}_{N} \cdot U^{\text{bwh}}_{C}}$.

By definition, we have $p = T_C(p^*)/T_{NC}(p^*)$ (ratio of the number of original bytes sent per sec with compression to the total number of bytes sent per sec). Since $T_C(p^*) = B \cdot \epsilon / r$ and $T_{NC}(p^*) = B \cdot (1 + \epsilon \cdot (1/r - 1))$, we get $p^* = \frac{1}{1 + r \cdot (1/\epsilon - 1)}$. Plugging in ϵ, we get Equation 7. □

3 DISCRIMINATIVE FINE-GRAINED MIXING

The main idea behind DFGM is to perform compression on only a subset of the attributes in the data stream and to adjust this subset dynamically as a function of the available computation and bandwidth resources.

The goal here is to avoid compressing tuple attributes that are less amenable to compression and/or are costlier to compress. By prioritizing the compression of attributes that can achieve a higher compression ratio, the bandwidth resources can be put into better use. Similarly, by prioritizing the compression of attributes that result in less costly compression, the computation resources can be put into better use.

There are a number of observations that motivate the applicability of this idea in practice. In particular, different attributes in a stream can have: (i) different compression ratios using the same compression algorithm; (ii) different compression costs using the same compression algorithm; (iii) different compression algorithms that provide the best compression; (iv) different compression algorithms that provide the cheapest compression.

Figure 1 shows the time it takes to compress a 64K block using different compression techniques on different data patterns. The data type used is a 4-byte integer. For data patterns, 'random' represents a series of integers that were uniformly chosen at random, 'randomXfixedY' represent a series where X random integers are followed by Y occurrences of a fixed integer, 'consecutive' represents integers increasing by a fixed delta, and 'fixed' represent repeated occurrences of a fixed integer. For compression algorithms, 'zlib' and 'gzip' are two well known compressors, 'sameValComp' is a special-purpose simple compressor optimized for compressing sequences containing large segments of repeated values, and 'seqComp' is a similar compressor that is optimized for compressing sequences of values with a fixed numerical difference between them. It can compress integral numbers or even strings that contain a fixed prefix and an increasing sequence id. Since data streams are typed (each stream has a schema and each attribute has a type that is known at compile-time), building such special purpose compressors is possible.

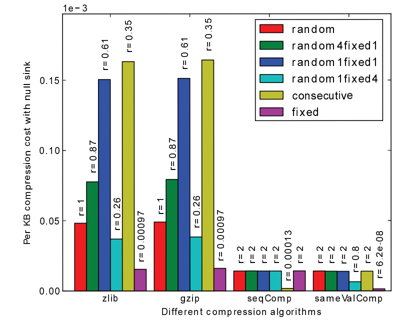


Fig. 1: Compression cost and ratio for different algorithms on different data patterns.

We observe from Figure 1 that for different data patterns, different compression algorithms provide the best results (w.r.t. compression ratio and cost), such as 'sameValComp' for 'fixed' and 'seqComp' for 'consecutive'. We see that special purpose compressors can achieve good compression with small cost, but only for the right data pattern. As for general purpose compression algorithms, it is important to note that the cost of compression is dependent on the data pattern, which further motivates the need for applying DFGM.

In stream processing applications, there is ample opportunity for DFGM. For instance, many data streams contain sequence numbers (usually 64-bit integers) that increment by one, date-time strings or time counters that are repeated (since data streams are generally time-ordered series), and categorical attributes with small domain sizes (such as the type of a financial transaction). Many of these attributes can provide good compression ratio, but even more importantly, in a very computationally inexpensive way if a data-specific compressor is used. Thus, we pick 'seqComp' and 'sameValComp' as example domain-specific compressors for this work.

Even in the absence of opportunities for effective and cheap compression, DFGM is still expected to provide improvement in throughput. This is because general purpose compressors have varying costs across different data patterns. We use 'zlib' and 'gzip' as examples, since they are well known and commonly available.

3.1 Formalization

We now formalize the DFGM problem. Let $A = \{a_i : 0 \le i \le |A|\}$ denote the list of attributes in a tuple for the data stream. For each attribute $a \in A$, we define:

- $r(a)$: the compression ratio for attribute a,
- $c_c(a)$: the compression cost for attribute a, and
- $s(a)$: relative size of attribute a in the tuple.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Here, $s(a) \in (0, 1]$ represents the ratio of the size of the attribute to the tuple size. All of the above are measured variables. We also define a set of decision variables:

- $V(a)$: 1 if attribute a is compressed, 0 otherwise

We define the optimization problem for DFGM as:

$$\text{argmax}_V \min\left(\frac{C}{c(V)}, \frac{B}{b(V)}\right), \tag{8}$$

where $c(V)$ is the per-byte computation cost and $b(V)$ is the per-byte bandwidth consumption. We have:

$$c(V) = \sum_{a \in A} s(a) \cdot \big(V(a) \cdot (c_p + c_c(a) + r(a) \cdot c_s) + (1 - V(a)) \cdot (c_p + c_s)\big) \tag{9}$$

$$b(V) = \sum_{a \in A} s(a) \cdot \big(V(a) \cdot r(a) + (1 - V(a))\big) \tag{10}$$

In Equation 9, for each attribute a, we are summing the cost of processing the attribute with compression (multiplied with $V(a)$, thus only contributes when the attribute is set to be compressed) and the cost without compression (multiplied with $1 - V(a)$) and scale the result with $s(a)$ (since only that fraction of bytes are from this attribute). Similar logic is applied in Equation 10, for the bandwidth consumption.

3.2 Handling Discreteness

One problem with the formulation we have so far is that, due to the discrete nature of the number of attributes, it may not be possible to find a solution that could outperform the one from uniform FGM, with respect to throughput. For instance, if there is only a single attribute ($|A|$=1), there are only two options: all-compress or no-compress. We solve this problem by applying compression using the decision variables V, but only with probability $p^*(V)$. Here, the mixing ratio can be given as in Equation 7, with the exception of replacing r with $r(V)$ and c_c with $c_c(V)$. Here, $r(V)$ represents the overall compression ratio and $c_c(V)$ represents the overall compression cost, for a given set of attribute compression settings V. We have:

$$r(V) = b(V) \tag{11}$$

$$c_c(V) = \sum_{a \in A} s(a) \cdot V(a) \cdot c_c(a) \tag{12}$$

In Equations 11 and 12 the compression ratio and cost are computed as aggregates over all attributes, with appropriate scaling using the relative attribute sizes. The final problem can be stated as follows:

$$\text{argmax}_V \, T(p^*(V)) \tag{13}$$

Here, the throughput function $T(.)$ is from Equation 1, with r and c_c replaced with Equations 11 and 12, respectively. $p^*(V)$ is from Equation 7. With this formulation, DFGM completely generalizes uniform FGM.

A brute-force algorithm to solve Equation 13 takes a long time as the number of attributes reaches 10 or so, due to the combinatorial explosion of solutions (V). Since the optimization needs to be performed frequently, this is unacceptable and we look at heuristic approaches.

Algorithm 1: greedyCNP($A, s(.), r(.), c_p, c_c(.), c_s, Y_{CN}(.)$)

```
Data: A: tuple attributes, s: relative sizes, r: compression ratios, c_p:
      application cost, c_c: compression costs, c_s: submission cost, Y_CN:
      utility function to be used
V(a) ← 1, ∀_{a∈A}                         ▷ Reset all attributes to compress
L ← sort(A, Y_CN)                          ▷ L is a sorted (using Y_CN) list on A
for a ∈ L, in decreasing order do
    V(a) ← 0                               ▷ Set attribute a to compress
    L ← L \ a                              ▷ Remove a from the list
    if C/c(V) > B/b(V) then                ▷ Bottleneck is bandwidth
        V(a) ← 1                           ▷ Revert a to no compress
p*(V) ← computeP(A, s, r(V), c_p, c_c(V), c_s)   ▷ Use Eq. 7
```

3.3 Model-based Algorithms

Here, we assume that all non-decision variables can be measured on a continuous basis, such as the compression, submission, and application costs, as well as the computation and bandwidth availability. In other word, we strictly follow the model we have developed so far.

The algorithms we describe are heuristic in nature. The main idea is to start from no-compress or all-compress and gradually move to the other direction unless an infeasible solution is reached. For instance, if we start with the no compress ($\forall_{a \in A} V(a) = 0$) state, at each step we can pick one attribute a and set $V(a) = 1$ unless the computation becomes the bottleneck ($B/b(V) > C/c(V)$). We call this algorithm 'greedyNC'.

The reverse algorithm, called 'greedyCNP', starts from all-compress ($\forall_{a \in A} V(a) = 0$), and at each step picks one attribute a and sets $V(a) = 0$ unless bandwidth becomes the bottleneck ($C/c(V) > B/b(V)$). The pseudo-code for code the algorithm is given in Algorithm 1. Since the 'greedyCNP' algorithms stops at a configuration V for which the computation is still a bottleneck, Equation 7 is used to set the mixing ratio to $p = p^*(V)$, whereas in 'greedyNC' the mixing ratio p is set to 1.

In these greedy algorithms we need to use a heuristic metric to decide the order in which the attributes are tried. For this purpose, we define a utility function, denoted by Y_{NC} for 'greedyNC', and $Y_{NC} = 1/Y_{CN}$ for 'greedyCNP'. For $Y_{NC}(a)$, we define a few alternatives:

- *LR*, lowest compression ratio: $1/r(a)$.
- *HB*, highest bandwidth used: $s(a) \cdot r(a)$.
- *SC*, smallest computation cost: $\frac{1}{s(a) \cdot (c_p + c_c(a) + r(a) \cdot c_s)}$.
- *HBC*, highest bandwidth gained per computation cost incurred: $\frac{1 - r(a)}{c_c(a) - (1 - r(a)) \cdot c_s}$.

To pick the next attribute to compress, we can locate the one that compresses well (LR), uses up the highest bandwidth (HB), incurs the smallest computation cost (SC), or provides the highest reduction in the amount of bandwidth used for unit of additional computation incurred when compressed (HBC).

Example: 'greedyCNP'. Consider the following setup. We have a stream with 4 attributes, $[a_1, \cdots, a_4]$. Assume that the compression ratios are $[0.25, 0.6, 0.9, 0.5]$, the relative sizes are $[0.16, 0.2, 0.4, 0.24]$, and the compression costs are $[10, 15, 25, 5]$. Further assume that the processing cost is 20 and the submission cost is 2. Finally, assume that the total computational capacity is 150 and the bandwidth capacity is 4. Based on these setting,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

the list L that contains the attributes ordered by the metric Y_{CN} based on HBC heuristic is computed as $[a_3, a_2, a_1, a_4]$. This means that the 'greedyCNP' algorithms will consider the attributes for which to turn off compression in this order.

Initially, the 'greedyCNP' algorithm will set $V(a_i) = 1, \forall_{i \in [1..4]}$. That is, we start with all-compress. First we will consider turning off compression for a_3. After setting $V(a_3) = 0$, we still have $\frac{C}{c(V)} \leq \frac{B}{b(V)}$ (Cpu is still the bottleneck), as $5.52 \leq 7.35$. Thus, we move to the next iteration. This time, we try turning off compression for a_2. This succeeds as well, since after setting $V(a_2) = 0$, we still have $\frac{C}{c(V)} \leq \frac{B}{b(V)}$, as $6.17 \leq 6.58$. Next, we try turning off compression for a_1. However, setting $V(a_1) = 0$ results in $\frac{C}{c(V)} > \frac{B}{b(V)}$ (bandwidth becomes the bottleneck), as $6.42 > 5.68$. As a result, we leave $V(a_1) = 1$. Finally, we try a_4 and similar to the case for a_1, this fails due to bandwidth becoming the bottleneck. At the end, we get $V = [1, 0, 0, 1]$.

After finalizing V, we need to set the mixing ratio $p^*(V)$. We have $r(V) = 0.76$ and $c_c(V) = 2.8$. This implies that DFGM for the computed V is similar to having a uniform compression algorithm with compression ratio 0.76 and compression cost 2.8. Finally, applying Equation 7, we get $p^*(V) = 0.84$.

3.4 Online Algorithm

As we discussed earlier, in practice it is a challenge to measure all the model variables on a continuous basis. As such, we now look at an online algorithm that relies on three easily measurable runtime metrics, namely:

- Overload (denoted by o) is a Boolean metric that determines whether the Cpu is fully utilized.
- Congestion (denoted by g) is a Boolean metric that determines whether the network is fully utilized.
- Throughput (denoted by t) is a metric that measures the rate at which the tuples are being processed.

The overload metric can be measured using Cpu utilization, through OS APIs available in most operating systems. The congestion metric can be measured by looking at the size of the network buffers and if that is not available at the application level, the congestion can be measured using blocking I/O on sends and measuring the blocking time[2].

The online algorithm works in periods. It observes the throughput, overload, and congestion for some time, called the *adaptation period*, and then adjusts the compression decisions based on these values.

Here we describe one such algorithm that works on the following principles:

- *Contract*. Turn compression on for an additional attribute if there is congestion but no overload, unless we have been there but seen less throughput.
- *Expand*. Turn compression off for an attribute if there is no congestion but overload, unless we have been there before but seen less throughput.
- *Revert*. Go back to the previous setting if throughput decreases due to Contract of Expand after an adaptation period has passed.

2. InfoSphere Streams [11] middleware uses this latter approach to come up with a metric called "congestion index".

Algorithm 2: onlineDFGM(g, o, t)

```
Data: g: congested?, o: overloaded?, t: throughput
i ← |a ∈ A : V(a) = 1|                        ▷ Compressed attribute count
if t≺ > t and a≺ ≠ nil then                    ▷ Throughput decreased
    V(a≺) ← 1 − V(a≺)                          ▷ Revert back the last decision
else                          ▷ There may be a chance to improve throughput
    a≺ ← nil                                   ▷ Set last action taken to none
    if g and ¬o then                           ▷ Congested but not overloaded
        if i < |A| and T_{i+1} ≥ t then        ▷ Open from above
            a≺ ← argmax_{a∈A:V(a)=0} Y_NC(a)
            V(a≺) ← 1                          ▷ comp=on for next best attrb.
    else if ¬g and o then                      ▷ Not congested but overloaded
        if i > 0 and T_{i−1} ≥ t then          ▷ Open from below
            a≺ ← argmin_{a∈A:V(a)=1} Y_NC(a)
            V(a≺) ← 0                          ▷ comp=off for next best attrb.
T_i ← t                                        ▷ Remember the performance at level i
t≺ ← t                                         ▷ Remember the last throughput
```

The pseudo-code for the 'onlineDFGM' algorithm that implements this logic is given in Algorithm 2. The algorithm maintains the following three variables across adaptation steps:

- T_i: throughput observed at level i (the number of attributes compressed), initialized to ∞ at start-up,
- $t^{\prec}$: throughput observed at the end of the previous adaptation period, initialized to $-\infty$, and
- $a^{\prec}$: the attribute whose compression setting was changed at the end of the previous adaptation period, initialized to *nil*.

The algorithm simply applies the Contract, Expand, and the Revert principles using the utility function $Y_{NC}(.)$ to determine the next attribute for which the compression will be turned on/off. The T_i values are used to avoid oscillation as part of the Contract and Expand principles, whereas the $t^{\prec}$ and $a^{\prec}$ values are used to implement the Revert principle.

This version of the 'onlineDFGM' algorithm has a serious flaw: it cannot handle changes in the availability of the computation capacity or bandwidth capacity. For instance, assume that in the steady state we are compressing two attributes and compressing one more results in computation becoming the bottleneck and the throughput going down. Further assume that after some time the computation capacity available to us has increased, so it is possible to compress one more attribute. However, due to the $T_{i+1} \geq t$ check, we won't be able to re-explore this setting. One solution to these adaptivity problems is to periodically reset the T_i values back to ∞ in order to let the algorithm re-explore (similar to [24]). This variation of the algorithm can adapt to changes, but the reset interval should be kept large to avoid oscillation, and thus the adaptation cannot happen at small time-scales. Also, unlike the model based algorithms, the online algorithm suffers from discreteness problem.

Example. We continue using the example setup from Section 3.3. With the online algorithm, the list of attributes are considered in reverse order, $[a_4, a_1, a_2, a_3]$, since we start from the no-compress setting. Initially, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Algorithm 3: adaptiveDFGM(Q)

```
Data: Q: the buffer of tuples
while not terminated do                                  ▷ Thread's main loop
    Wait until Q has tuples                              ▷ Until data arrives
    Let L_s = block of tuples at the front of Q
    Try sending L_s without blocking                     ▷ Non-blocking I/O
    if would block then                                  ▷ Compress more
        c ← 0                                            ▷ Amount compressed
        for each block L of tuples in Q do
            Let A' = {a ∈ A : a is not compressed in L}
            if L ≠ ∅ then                                ▷ Can compress further
                a ← argmax_{a∈A'} Y_NC(a)                ▷ Best attribute
                Compress attribute a in L
                c ← c + s(a)                             ▷ Update amount compressed
                if c ≥ |L_s| then break                  ▷ A block's worth
    else
        Dequeue L_s from Q                               ▷ Done sending this block
```

will observe congestion, since ($\frac{C}{c(V)} > \frac{B}{b(V)}$, as $6.82 > 5$). Since there is no knowledge about a higher level (open from above), the online algorithm will compress a_4 next. The congestion will persist ($\frac{C}{c(V)} > \frac{B}{b(V)}$, as $6.53 > 5.68$). Since the throughput has increased ($5.68 > 5$), the algorithm will not revert back. And since we do not have knowledge about a higher compression level, the next attribute in line, a_1 will be compressed. This time, we will observe overload ($\frac{C}{c(V)} \leq \frac{B}{b(V)}$, as $6.17 \leq 6.58$). The algorithm will check if there is a need to revert back. Since the throughput has increased ($6.17 > 5.68$), this won't be attempted. Next it will check if overload can be resolved by reducing the compression level. However, since it is known that the level below provides less throughput, the algorithm will settle down.

3.5 Online, Fine-grained Adaptive Algorithm

We now look at an algorithm that is both online and adaptive. Interestingly, it does not use metrics directly, but it indirectly relies on the bandwidth and computation capacity availability. Here we describe the main operation logic of the algorithm in general terms and provide the intuition for its adaptation properties. In the next section, we look at various implementation issues.

We assume that there is a transport thread that picks up tuples to submit from a buffer that is shared with the application level thread(s) that enqueue the tuples into this same buffer. The pseude-code for the logic executed by the transport thread is given in Algorithm 3.

The transport thread takes a block of tuples from the buffer and tries sending it using *non-blocking* I/O. If the block is submitted in full, the algorithm moves on to executing the same logic for the next block of tuples. Otherwise, the algorithm tries to compress one block's worth of data, but it does this 'vertically'. For each tuple block in the buffer, from the oldest towards the newest, it compresses one attribute per block until the total amount of data compressed is equal to the size of a block. This means that the algorithm keeps track of the number of attributes compressed for each tuple block. The order in which the next attribute to compress is determined by the utility function $Y_{NC}(.)$.

When neither the bandwidth nor the computation is the bottleneck for all-compress and for no-compress (i.e., workload is not sufficient to utilize all resources), the algorithm will send all tuples without compression since all submissions will go through in the first try.

When the bandwidth is the bottleneck but computation is not for all-compress (Table 1, row 1), the algorithm will compress all tuples. This is because the tuples will build up in the buffer when the incomplete submissions happen frequently due to bandwidth unavailability. In response, the algorithm will start compressing tuples attribute-by-attribute until bandwidth is available. But even with partially compressed tuples, the bandwidth is still the bottleneck, and thus the build-up will continue. Eventually all sent tuples would be fully compressed.

When the computation is the bottleneck but bandwidth is not for no-compress (Table 1, row 2), the algorithm will not compress any tuples. Again this is because all submissions will go through in the first try.

The true benefit of the algorithm compared to uniform mixing is when the computation is the bottleneck for all-compress and the bandwidth is the bottleneck for no-compress (row 3 in Table 1). In this case, the algorithm will perform partial compression, preferring to compress attributes that are cheaper to compress and compress well, based on the utility function.

The value of the utility function $Y_{NC}(a)$ for each attribute a is determined by online profiling. In particular, every *profiling period*, a block of tuples is analyzed to determine the compression cost, ratio, and the relative attribute size. Furthermore, the contents are analyzed to determine if custom compressors are applicable. The latter can also be obtained from the compiler without the need for profiling if they can be derived from the semantics of the stream processing language at hand or through user hints. It is expected that the utility function values for attributes do not change frequently and thus profiling does not need to be performed frequently.

4 Implementation

We now describe our implementation of the adaptive algorithm. In particular, we look at the practical considerations that has to be taken into account when implementing Algorithm 3.

Figure 2 provides a depiction of the operational state of the algorithm. As outlined earlier, the algorithm is implemented by having a buffer in between the application and the network. This buffer is called the *compression buffer* (outermost box in the figure). Recall that the application threads enqueue tuples into the compression buffer. The goal of the transport thread is to submit these tuples to the network, and opportunistically compress data when bandwidth is not available.

In our implementation, the compression buffer has a two-segmented structure. The first segment, called the *tuple buffer*, keeps the enqueued tuples. The second segment, called the *block buffer*, keeps the enqueued tuples divided into *block*s. Each block contains the wire representation of the list of tuples associated with it as well. The wire representation is the result of serializing the tuples on an attribute-by-attribute basis.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

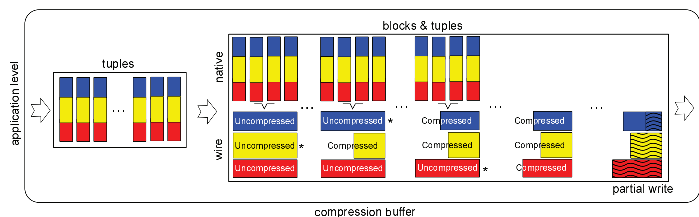


Fig. 2: Operational state of the online, adaptive algorithm

Since DFGM uses attribute-based compression, it needs to accumulate sufficient number of tuples to achieve reasonable compression ratios for each attribute. The block size H should be set such that $r_H(a) < (1-\sigma)\cdot r_\infty(a)$, where $r_H(a)$ is the compression ratio that can be achieved with a block size of H and σ is a small number, typically less than 0.1. However, the block size may also impact the latency. The acceptable latency is highly dependent on the application's quality-of-service (QoS) requirements. Given the average tuple size, the latency introduced due to a block can be computed by the number of tuples in a block times the inverse of the stream rate achieved. In the figure, a block keeps 4 tuples (this is a rather small block used for illustration purposes only). In the evaluation part we study the impact of buffer and block sizes on performance.

Since the application threads may generate tuples at a higher rate than the transport layer can handle, the compression buffer has an upper bound on its size. The buffer size refers to the total number of tuples in the compression buffer, including the tuple and the block buffers. The transport thread is responsible for moving tuples from the tuple buffer into the block buffer. At each iteration, it moves one block's worth of tuples (if exists) and attempts to submit the oldest block to the network. If the submission results being incomplete (using non-blocking I/O call), then the transport thread attempts to perform compression on the blocks, starting from the oldest, moving towards the newest. It compresses one block's worth of data using partial compression: the next attribute in line is compressed for each block considered.

In the figure, we could see that the oldest block has all its attributes compressed, whereas some newer ones have less attributes compressed. This is due to the fact that at each compression attempt, we do not compress a fixed number of blocks, but instead a fixed number of bytes. This is done to emulate the behavior of a static system, where at each iteration a block is formed, compressed, and sent. Each block keeps a variable that points to the next attribute to be compressed. This is shown using the $\star$ sign in the figure. Note that the attributes are considered in the order of their utility. In the figure, this order is: yellow, blue, red. This is easy to observe, as going from left to right, the first compression we see is for the yellow attribute, the second is for the blue attribute, and the third is for the red attribute.

The reason original tuples are kept together with the wire-format blocks is that special-purpose compressors are templatized on data types. Given an attribute to compress and its type, the compressors iterate over the tuples and stream the compressed output into the proper location within the serialized block. Furthermore, for special-purpose compressors, the value of the attribute with its native in-memory layout is required for performing operations on it (e.g., subtraction for the 'seqComp' compressor). To minimize the overhead of memory allocation and data copying, we perform the compression in-place, by overwriting the wire-formatted data. The original tuples can be discarded if and when all attributes are compressed. In the figure, tuples associated with the oldest two blocks are already discarded.

Wire-formatted blocks contain data in the *column-oriented* format, where the values of the same attribute from subsequent tuples are placed consecutively in the serialization. Since we perform compression on an attribute-by-attribute basis, the compression leaves a gap in the serialization as we do not want to pay the cost of shifting the serialized representations of the rest of the attributes. These gaps can be seen in the figure as part of the blocks that have compressed attributes. As a result, we send the serialized blocks to the network transport using *scattered I/O*. In particular, we use the `writev` call from the Standard C Library.

DFGM incurs some additional overhead due to the layout of the partially compressed serialized blocks. First, on the decompression side, we need to distinguish the sub-blocks corresponding to different attributes within a serialized block. For this purpose we include the size of the sub-blocks as part of the block header. This would require $4\cdot|A|$ bytes, where 4-byte integers are used to encode the size of each sub-block. However, for this purpose we use base 128 varint variable length encoding. This reduces the size to half, that is to $2\cdot|A|$ bytes, for most practical setups. Second, we need to identify whether each sub-block is compressed or not, which requires $|A|/8$ bytes using a single bit to represent the compression setting for each attribute.

Finally, a `writev` call in non-blocking mode can result in *partial write*s. In Algorithm 3 we assumed that the transport thread compresses attributes from the not yet sent tuple blocks when the send attempt returns 'would block'. In practice, such non-blocking calls may write partial data and then return indicating that further write would block. The figure illustrates this on the oldest block, where the write is shown to have sent the yellow and red attributes, but the blue attribute is sent partially. As a result, we only apply compression to the to be sent block if it has not been partially written, otherwise we start the compression from the next block available.

5 Evaluation

We evaluate the effectiveness of DFGM, using both model-based results that study a wide range of factors, as well as results that use our C++ implementation on real-world data sets. The model based experiments evaluate the impact of various factors on three important metrics, namely: the throughput achieved, the bandwidth and Cpu utilizations. The implementation

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

props.\names	seqNo	RIC	Date[G]	Time[G]	Type	Bid_Price	Bid_Size	Ask_Price	Ask_Size	Qualifiers
types	long	string	string	string	string	double	double	double	double	string
sizes	0.08	0.07	0.15	0.17	0.09	0.08	0.08	0.082	0.08	0.11
best alg.	seq	zlib	sameVal	sameVal	sameVal	zlib	zlib	zlib	zlib	sameVal
compr. ratios	~ 0	0.28	~ 0	0.27	0.13	0.25	0.1	0.25	0.11	0.46
compr. cost	0.006	0.224	0.006	0.011	0.012	0.245	0.114	0.243	0.121	0.019
compr. rank	0	7	1	3	2	9	5	8	6	4

TABLE 2: Properties of the attributes in the TAQ data set.

based experiments compare FGM and DFGM in terms of throughput and showcase the adaptivity of our solution by dynamically changing the bandwidth availability.

5.1 Experimental Setup

We describe the experimental setup for the model and implementation based experiments.

Description	*default*	*range*
# of tuple attributes	10	$[1, 20]$
attrb. size Zipf param.	0.2	$[0, 1]$
compr. ratio Normal mean	0.1	$[0.01, 1.2]$
compr. ratio Normal stddev	2.0	$[0, 2]$
available bandwidth	1Gbit/s	[100Mbit/s, 10Gbit/s]
available Cpu	1	$[0, 1]$
compr. cost scale	10	$[1, 20]$
computation cost scalers		
application	$1\times$	—
compression	$2\times$	$[1, 10]$
submission	$0.1\times$	—

TABLE 3: Experimental parameters: default values and ranges

Model parameters: Table 3 shows the list of model parameters used. Here we describe the parameter settings that are not immediately obvious from the table. The relative attribute sizes are generated using a Zipf distribution, where attribute a_i has size proportional to $1/(1+i)^\alpha$, where α is the Zipf parameter. The compression ratios are picked using a Normal distribution with mean μ and standard deviation σ, but the distribution is clipped to fit the range $[0.01, 1]$. For $\mu = 0.1$ and $\sigma = 2$ (default), we have a mean compression ratio of 0.5. For smaller values of the σ, the mean gets closer to μ. The available bandwidth is set to a default value of 1Gbit/sec. The Cpu availability is set to 1 by default. We adjust the processing costs such that it is possible to process tuples at $5\times$ the rate of the default bandwidth when there is no compression or tuple submission and all Cpu is available. The relative costs of application, compression, and submission costs are given in the table. The compression cost scale is the relative cost of compression for the best compressing attribute to that of the worst compressing attribute. Here we assumed a linear relationship between costs and the compression ratio.

Real-world data-sets: We use a financial data stream called TAQ [5] as our main workload. The data is a sequence of trade and quote transactions, where trade transactions are characterized by the price of an individual security and the number of securities that were acquired/sold (i.e., volume). The quote transactions can either be a bid or an ask quote. A bid quote refers to the price a market maker will pay to purchase a number of securities and an ask quote refers to the price a market maker will sell a number of securities for.

Table 2 provides the properties of the attributes found in the TAQ stream. In particular, we provide the types of the attributes, their relative sizes, the best compression algorithm (based on $(1 - r(a))/c_c(a)$) for the attribute, the compression ratio, normalized compression cost, and finally the rank of the attribute for compression (0 meaning the attribute is the first one to be compressed).

We use two additional workloads. One is from the Linear Road Benchmark [7]. This dataset, referred to as the LinearRoad dataset, contains location (road, segment, direction, etc.) and time information about cars driving on a simulated highway. In this workload, all attributes are numerical (a total of 10 attributes) and have similar size. The characteristics of the attributes with respect to compression is not as diverse as the TAQ workload. We expect lesser benefit from discriminative mixing for this dataset. The other workload we use is from a network monitoring application (used in [25]) that monitors Linux log files for login attempts. This dataset, referred to as the LogWatch dataset, has 7 diverse attributes, but interestingly one of the attributes has large size, constituting a majority of the tuple's content.

Experimental system: For experiments, we used two machines, each with a 2.2GHz Intel processor that has 32KB L1 data, 32KB L1 instruction, and 256 KB L2 cache per core, 6 MB L3 cache that is shared for all cores, and 4GB of memory. The processor has 4 cores, but we only use one core for the transport thread. We used a 1Gbit Ethernet network for the communication. The OS used was FreeBSD 9.

For controlling the bandwidth available for communication, we used the `ipfw` command line tool available on BSD-based Unix systems. In particular, we used the dummynet traffic shaper facilities to set the bandwidth of the connection to the desired value.

5.2 Model based experiments

We discuss the set of experiments conducted using our model, based on the parameters listed in Table 3.

Impact of Cpu availability: Figure 3 plots throughput as a function of the Cpu availability, for different approaches. Here, the goal is to show the superiority of discriminative mixing over uniform mixing. The 'pOnly' approach represents uniform mixing. 'subsP' represents the optimal discriminative mixing, with $p^*(V)$ used to handle the discreteness problem. It tries every possible subset to find the best setting of V in terms of throughput. 'subsD' is similar, but does not use p^*. 'plain' represents no-compress and 'comp' represents all-compress. Results are relative to the throughput of the 'pOnly' approach.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

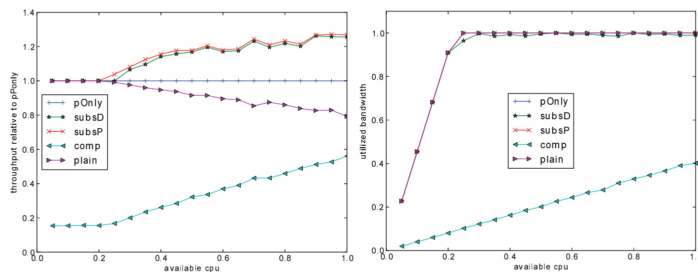


Fig. 3: Relative throughput as a function of Cpu availability.

Fig. 4: Bandwidth utilization as a function of Cpu availability.

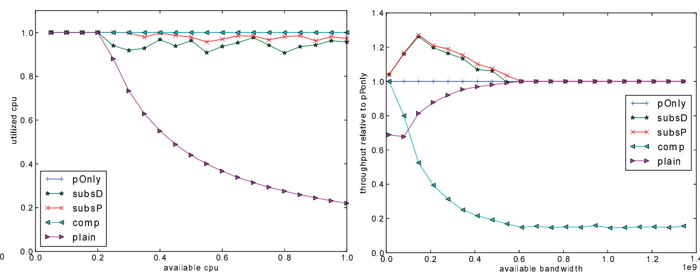


Fig. 5: Cpu utilization as a function of Cpu availability.

Fig. 6: Relative tputàs a function of bwidth. availability.

We observe from Figure 3 that for low Cpu availability all approaches except 'comp' achieves the same throughput. As more Cpu becomes available, the bandwidth becomes the bottleneck and the 'comp' approach starts to gain in terms of relative performance (as it compresses data) and 'plain' starts to lose its relative effectiveness (as it does not perform compression). More importantly, we see that discriminative mixing reaches up to 26% higher throughput compared to uniform mixing. The throughput difference between 'subsD' and 'subsP' is small, as we use 10 attributes by default.

Figure 4 plots the utilization of the bandwidth as a function of the Cpu availability for different approaches. We see that all approaches, except 'comp', are able to saturate the bandwidth starting from modest values of the Cpu availability (> 0.25). However, as it was clear from Figure 3, discriminative mixing is able to make the most out of fully utilizing the bandwidth as it is able to compress as much as possible, without making Cpu the bottleneck (as opposed to 'comp').

Figure 5 plots the utilization of the Cpu as a function of the Cpu availability. We see that all approaches, except 'plain', are able to saturate the cpu. 'subsD' is able to achieve slightly lower cpu utilization compared to 'subsP', due to the discreteness issue. 'plain' suffers significantly as it does not perform compression and hits the bandwidth limit early.

Impact of bandwidth availability: Figure 6 plots throughput as a function of the bandwidth availability, for different approaches. Again, the goal is to show the superiority of discriminative mixing over uniform mixing. We see that when the bandwidth is plenty all approaches except 'comp' reach the same throughput. 'comp' suffers relatively as it hits the Cpu bottleneck. For very low bandwidth availability (close to 0), all approaches except 'plain' reduce to all compress, so they all provide the same throughput, except 'plain' which suffers from lack of compression. Most importantly, we observe that discriminative mixing provides up to 30% higher throughput compared to uniform mixing. Figure 7 plots the utilization of the bandwidth as a function of the bandwidth availability for different approaches. We see that all approaches, except 'comp', are able to saturate the bandwidth until the cpu becomes the bottleneck ($0.6 \cdot 10^9$, at which point all approaches except 'comp' reduce to no compress), after which point additional bandwidth ends up being unused. However, as it was

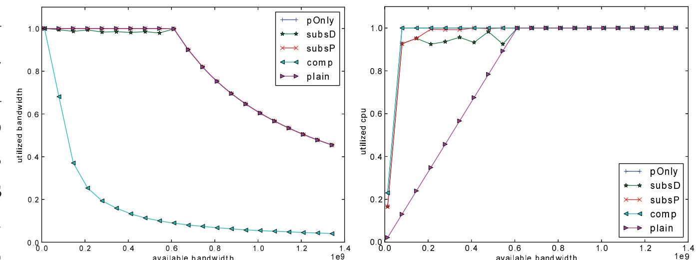


Fig. 7: Bandwidth utilization as a function of bandwidth availability.

Fig. 8: Cpu utilization as a function of bandwidth availability.

clear from Figure 6, discriminative mixing is able to make the most out of fully utilizing the bandwidth as it is able to compress as much as possible, without making Cpu the bottleneck.

Figure 8 plots the utilization of the Cpu as a function of the bandwidth availability. We see that all approaches, except 'plain', are able to saturate the Cpu quickly as the bandwidth availability increases ($> 0.8 \cdot 10^8$). Again, 'subsD' is able to achieve slightly lower Cpu utilization compared to 'subsP', due to the discreteness issue. 'plain' suffers for low bandwidth availability since it does not compress data and as such cannot utilize the cpu. Once there is enough bandwidth availability, cpu becomes the bottleneck for all approaches.

Joint impact of Cpu and bandwidth availability: Figure 9 plots the available bandwidth as a function of both the bandwidth and the Cpu availability. Again the throughput is relative to that of 'pOnly'. As expected, when the bandwidth is plenty, all approaches except 'comp' provide the same throughput. When the bandwidth is scarce, all approaches except 'plain' provide the same throughput. The sweet spot for discriminative mixing is when both the Cpu and the bandwidth availability are low. This is the region where Cpu becomes the bottleneck for all compress, whereas bandwidth becomes the bottleneck for no compress, that is the same region we have identified in Table 1 for fine-grained mixing. We see that when there is opportunity for performing fine-grained mixing, doing it via discriminative mixing provides up to 30% better throughput compared to uniform mixing.

Impact of heuristic approaches: Figure 10 plots the throughput (absolute) as a function of the Cpu availability, for different heuristic approaches. Here, we use 'subsP' as the upper bound on the throughput and 'pOnly' (uniform mixing) as the baseline approach. There are a number of important observations from the figure.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

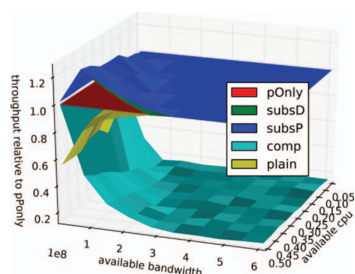


Fig. 9: Throughput vs. Cpu and bandwidth availability.

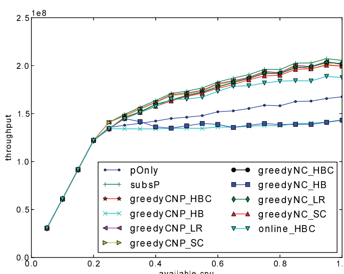


Fig. 10: Tput. as a function of Cpu, for different heuristics.

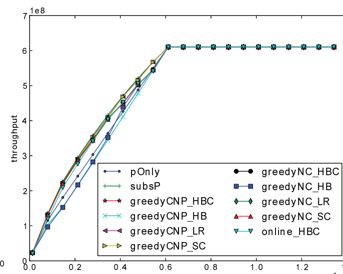


Fig. 11: Tput. as a function of bwidth., for different heuristics.

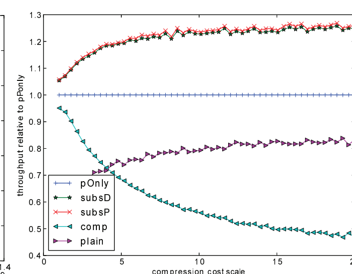


Fig. 12: Tput. as a function of compression cost scale.

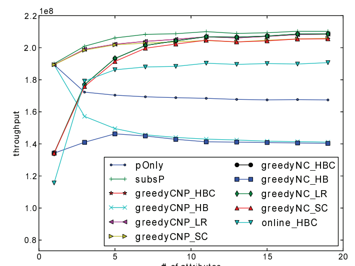


Fig. 13: Throughput as a function of # of attributes.

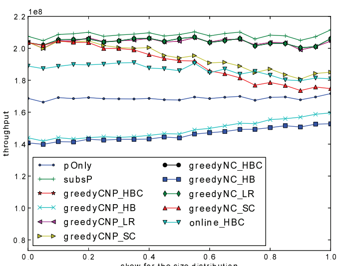


Fig. 14: Throughput as a function of attribute size skew.

First, all greedy algorithms perform very close to the optimal, with the exception of those that use the *HB* (highest bandwidth used) metric. Second, these greedy algorithms all perform better than the 'pOnly' baseline, providing up to 22% higher throughput. Third, the online algorithm that uses HBC metric, that is 'onlineHBC' also performs up to 15% better than the 'pOnly' baseline, yet the throughput it achieves is slightly below that of the greedy algorithms. Third, the 'CNP' variants of the greedy algorithms have a very small throughout advantage compared to the 'NC' variants due to applying probabilistic compression to solve the discreteness problem. We will study the impact of the number of attributes on this difference separately, as part of the sensitivity studies. Finally, we observe that increasing Cpu availability makes it possible to use compression to better utilize the bandwidth that becomes the bottleneck.

Figure 11 plots the throughput (absolute) as a function of the bandwidth availability, for different heuristic approaches. The results are similar in nature to those from Figure 10, with respect to the comparative performance of the difference approaches. One minor variation is that, the 'onlineHBC' approach performs closer to the greedy approaches compared to the Cpu graph from Figure 10. When the available bandwidth reaches a certain threshold, all approaches reduce to no compression and provide the same throughput. When the bandwidth is extremely scarce, all approaches reduce to all compress and again provide similar throughput. For low bandwidth availability scenarios fine-grained mixing (for all variations except *HB*-based greedy approaches) again outperforms uniform mixing.

Sensitivity to compression cost scale: Figure 12 studies the sensitivity of compression schemes to the compression cost scale, that is the relative compression cost of the least compressible data compared to the compression cost of the most compressible data. Recall that we use this to study the impact of custom compressors that can provide extremely cheap compression at a very low cost. The figure plots the throughput relative to that of the 'pOnly' approach. We see that 'comp's relative performance degrades as the relative cost of difficult to compress data increases. This is expected as the all-compress approach is wasting computational resources even more when the cost of compression increases more with reducing compression ratio. On the other hand, discriminative mixing excels when there are attributes for which compression can be done very effectively and very cheaply, as well as those for which compression is costly and ineffective. Discriminative mixing achieves this performance by prioritizing the attributes to compress and thus achieves higher throughput compared to 'pOnly'. When the cost of compression is the same irrespective of the compression ratio (point 1 on the x-axis), the throughout gained from applying discriminative mixing over uniform mixing is the least (5% for this particular setting), but as the costs decreases for better compressing attributes (such as due to using custom compressors), the gain in throughput increases significantly (up to $\%25$).

Sensitivity to number of attributes: Figure 13 studies the impact of the number of attributes on the effectiveness of the compression. It plots the absolute throughput as a function of the number of attributes. An important observation from the figure is that, the approaches that do not perform probabilistic compression, such as the 'greedyNC' variants, suffer when there is a single attribute, since they reduce to either all compress or no compress. Their performance catches up only when the number of attributes go over 5. Another important observation is that, the 'pOnly' and 'greedyCNP' approaches both perform optimally when there is only a single attribute. The performance of uniform mixing drops as the number of attributes increases, whereas the performance of discriminative mixing increases up to 8 attributes. To summarize, this experiment shows that discriminative mixing should be performed with probabilistic compression to avoid throughput sub-optimalities resulting from discreteness due to a few number of attributes.

Sensitivity to relative size distribution: Figure 14 plots the throughput as a function of the skew in the attribute size distribution. Recall that relative attribute sizes are picked using a Zipf distribution with parameter α. For

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

$\alpha = 0$, we have a uniform distribution, and as the α increases the distribution becomes more skewed. We observe that the uniform mixing is not impacted by the skew. Same is true for optimal discriminative mixing, that is 'subsP'. Among the heuristic approaches, the *SC* (smallest computation cost) based approaches show decreasing throughput as the skew increases. The *HB* (highest bandwidth) based approaches are again performing worse than uniform mixing, yet their throughput increase with increasing skew. Overall discriminative mixing holds a steady advantage over uniform mixing for the entire range of the skew parameter ($\sim 25\%$ for greedy approaches with *HBC* or *LR*).

Sensitivity to compression ratio distribution: Figure 15 plots the absolute throughput as a function of the standard deviation for the compression ratio. Recall that the compression ratios for the attributes are picked using a Normal distribution that is restricted to the range $[0, 1]$. As the standard deviation gets close to 2 (the right end of the x-axis), we have almost a uniform distribution, whereas as it reaches 0 (the left end), we have a fixed compression ratio of $\mu = 0.1$. The throughput decreases as the deviation increases, since the overall mean compression ratio increases due to the range clipping. When there is no variation in the compression ratios (left end), there is no additional benefit provided by discriminative mixing, since we also model the costs as relative to the compression ratio. As the variance in the compression rates increases, discriminative mixing starts providing improvement over uniform mixing. Interestingly, when the deviation of the compression ratio is low (around 0.3), the 'onlineHBC' algorithm starts performing worse than the 'pOnly' approach, where all greedy algorithms except *HB* variants outperform the 'pOnly' approach.

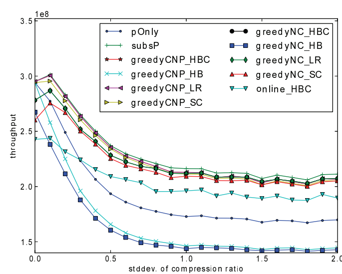


Fig. 15: Tput. as a function of compression ratio deviation.

Summary: We have shown that DFGM can be more effective compared to uniform FGM (up to 30%). When model parameters are available at run-time, discriminate mixing can be implemented cheaply using the greedy algorithms we introducted. The *HBC* (highest bandwidth gained per computation cost incurred) metric is the most robust one to be used with the greedy algorithms. The 'greedyCNP' performs better than 'greedyNC' for cases where the number of attributes is small. As such 'greedyCNP_HBC' is the most robust model-based algorithm. For cases where model parameters are not available at run-time, the 'online' algorithm still outperforms the optimal uniform mixing under most scenarios.

5.3 Implementation based experiments

We now present results on implementation based experiments, comparing the *adaptive and online* implementations of the fine-grained and uniform mixing approaches, based on the discussion given in Section 3.5.

Comparison of discriminative and uniform mixing: Figure 16 plots the absolute throughput as a function of the available bandwidth. For this experiment we compare the no compress, all compress, uniform mixing, and discriminative mixing approaches. We observe that the compress all performs well only for very low bandwidth scenarios, whereas compress none performs well only when the bandwidth is plenty. More interestingly, we observe that discriminative mixing is able to reach its maximum throughput earlier than uniform mixing. Uniform mixing cannot reach this maximum throughput when the block size is set to its default value of 4K. For this reason, we have also included the line for uniform mixing for a block size of 1K. With this setting, uniform mixing eventually reaches the maximum throughput but this happens only when the available bandwidth reaches close its maximum value of 1Gbit/sec. When the bandwidth availability is at 500Mbits/s, we see that discriminative mixing with 4K blocks provides around 40% improvement over uniform mixing with 1K blocks and around 18% improvement over uniform mixing with 4K blocks. Uniform mixing suffers from large blocks since making compression decisions on large blocks brings it closer to an approach that switches between no compress and all compress (looses its fine-grained nature). On the other hand, small blocks reduce the effectiveness of the compression. For discriminative mixing, even with larger blocks we can perform partial compression, which is an important advantage over uniform mixing.

Figure 17 plots the throughput for the LinearRoad dataset. Recall that this dataset has small variability among the characteristics of the attributes in terms of their compression cost, compression ratio, and data size. As such, discriminative mixing provides minor improvement over uniform mixing.

Figure 18 plots the throughput for the LogWatch dataset. Here, discriminative mixing shows benefit but only after the bandwidth availability reaches a certain threshold. Recall that this dataset has one attribute that constitutes majority of the tuple's content. As long as that attribute is one of the compressed attributes, discriminative and uniform mixing are not too different. Once discriminative mixing decides to exclude this attribute from compression (after there is sufficient bandwidth availability), it gains the throughput advantage.

Impact of block size on throughput: Figure 19 plots the absolute throughput as a function of the block size. We vary the block size between 1K and 64K. We observe that the maximum throughput that can be achieved increases with increasing block size, but eventually it converges to a fixed maximum. In particular, having a block size greater than 32K does not provide any additional benefit.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

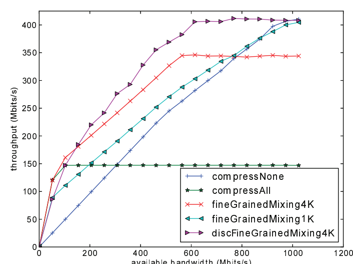


Fig. 16: Tput. as a function of available bwidth. – TAQ

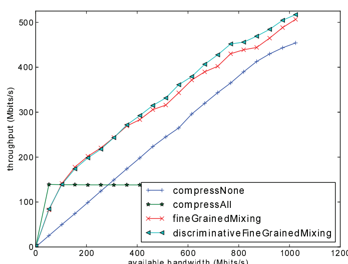


Fig. 17: Tput. as a function of available bwidth. – LinearRoad

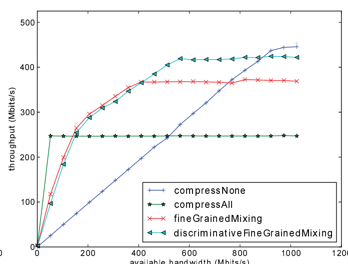


Fig. 18: Tput. as a function of available bwidth. – LogWatch

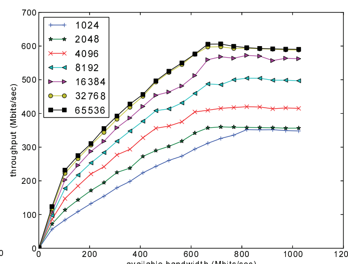


Fig. 19: DFGM throughput with different block sizes. – TAQ

A 32K block provides around 50% higher bandwidth compared to our default block size of $4K$. In general, a larger block size is able to reach a given throughput level at a lower bandwidth availability compared to smaller block size. However, a larger block size also implies larger latency, tolerance to which is highly dependent on the application requirements.

Adaptivity to bandwidth availability: Figure 20 plots the throughput achieved by discriminative fine-grained mixing and the available bandwidth as a function of time. For this experiment, we have changed the available bandwidth based on a step function. We use three different steps. The first step models low bandwidth availability, for which the bandwidth is the bottleneck and thus Cpu can be used to perform compression and achieve a throughput value that is higher than the available bandwidth. The first and the fourth segments in the figure show this, where the throughput is higher than the bandwidth. The second step models high bandwidth availability, for which the Cpu is the bottleneck. As a result, no compression is performed and the throughput achieved is lower then the available bandwidth. The second segment in the figure illustrates this. The third step models the scenario where the available Cpu and bandwidth resources are balanced, and thus the throughput achieved is close to the bandwidth available. The third segment in the figure shows this. Overall, the discriminative mixing is able to adapt well to bandwidth availability. Due to the fine-grained nature of the mixing and the non-blocking I/O based implementation, the adaptation is quick.

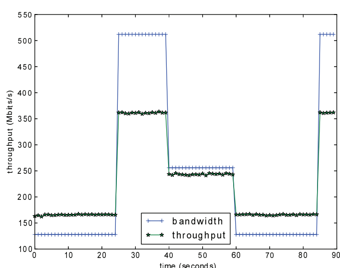


Fig. 20: Throughput and bandwidth availability.

6 Related Work

Data compression has been used in distributed systems to reduce the demand for network and disk bandwidth, disk space, and to address the disparity between I/O and processing speeds [9]. As such, it is natural to use adaptive compression to address variable bandwidth and Cpu availability in stream processing systems.

Our work is motivated by two lines of research. First is the work on adaptive fine-grained mixing by Pu and Singaravelu [20]. Fine-grained mixing switches between compression and no compression, at the granularity of individual blocks, and thus achieves partial compression. In this work, we model fine-grained mixing and provide a formula for the optimal mixing ratio. We then extend our model to discriminative fine-grained mixing, in order to take advantage of the structured nature of data streams as well as the significantly different compression ratios and costs of different stream attributes. We show that discriminative fine-grained mixing provides higher throughput compared to uniform mixing.

Several other works exist in the area of adaptive compression. In [13], a backward compatible version of the original fine-grained mixing algorithm is presented, which provides better compression ratios, wider range of data reduction and Cpu cost options, and parallelization strategies. The idea of using different compression approaches based on network and Cpu availability has appeared in several previous works, although without fine-grained adaptation. For instance, Dynamic Compression Format Selection (DCFS) [18] minimizes the total delay of transmitting and decompressing Java .jar files for remote execution. NCTCSys [19] system senses network and server parameters to efficiently use an appropriate method to balance the load and performance of the server and network for transmission of text files. Similar approaches for general data also exist [30], which monitor current network and processor resources, assess compression effectiveness, and automatically choose the best compression technique to apply. Finally, the Adaptive Compression Environment (ACE) [27] adopts a strategy that determines whether to use compression or not based on the Network Weather Service [31].

The second motivation for our work is compression in databases [8], [22], which is used not only to reduce disk space and to minimize disk I/O, but also to speed up query processing [21], [12]. Particularly relevant to our work is the use of compression in column-oriented databases [2], for which repeated attribute values are shown to be common and thus column-wise compression very effective [3]. Previous work on databases has also investigated the selection of appropriate compression methods to best exploit the Cpu and I/O bandwidth trade-offs for table scans [15].

Compression on streaming data has also been a popular technique for audio and video transmission, such

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

as Mpeg-1 layer 3 [16] for audio streaming. As another example, On-Demand Dynamic Distillation method [10] uses a proxy-based approach to tailoring content for clients, which is useful when images and video are transmitted to hardware constrained clients. However, these approaches typically use lossy compression techniques, and as such are not applicable in our setting.

7 Future Work

We consider two lines of future work. First is the investigation of using more than one thread for compressing the buffered data. This has been studied to some extent in the context of uniform fine-grained mixing [13]. A related issue is the use of compression algorithms that have built-in support for parallelism [4], [17].

Second, we would like to extend our model to cover the receiving end of the system (where partial decompression is done). When the receiver processing is heavy (due to application logic) or decompression takes more time than compression (asymmetrical algorithms [23]), this will impact the optimal mixing ratio.

8 Conclusion

We introduced an adaptive compression scheme for data stream processing systems, called discriminative fine-grained mixing (DFGM). We rely on the typed and structured nature of the data streams to select an effective subset of attributes to compress, in order to best utilize the available bandwidth without making the Cpu a bottleneck. When the computational resources are not sufficient to compress the entire stream, our approach judiciously selects the attributes that can bring good reduction in the used bandwidth at a low computational cost. Furthermore, the algorithm can quickly adapt as the bandwidth, Cpu, and workload availability changes. Through a detailed experimental evaluation, we have shown that DFGM outperforms uniform mixing, across a wide-range of values for the system parameters.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented database systems. *VLDB Journal*, 2(2), 2009.

[3] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD*, 2006.

[4] M. Adler. pigz - parallel gzip. http://www.zlib.net/pigz/. retrieved Jan 2012.

[5] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Processing high data rate streams in System S. *Elsevier JPDC*, 71(2), 2011.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[7] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A stream data management benchmark. In *VLDB*, 2004.

[8] G. V. Cormack. Data compression on a database system. *CACM*, 28(12), 1985.

[9] F. Douglis. On the role of compression in distributed systems. *ACM Operating Systems Review*, 27(2), 1993.

[10] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *ASPLOS*, 1996.

[11] B. Gedik and H. Andrade. A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere Streams. *Software: Practice and Experience*, 42(11), 2013.

[12] G. Graefe and L. D. Shapiro. Data compression and database performance. In *ACM SAC*, 1991.

[13] M. Gray, P. Peterson, and P. Reiher. Scaling down off-the-shelf data compression: Backwards-compatible fine-grain mixing. In *IEEE ICDCS*, 2012.

[14] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. SPL language spec. Technical Report RC24897, IBM, 2009.

[15] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *ACM SIGMOD*, 2007.

[16] ISO. Information technology – Coding of moving pictures and associated audio for digital storage media – Part 3: Audio. Technical Report ISO/IEC 11172-3, ISO, 1993.

[17] S. T. Klein and Y. Wiseman. Parallel Lempel-Ziv coding. *Elsevier Discrete Applied Mathematics*, 146(2), 2005.

[18] C. Krintz and B. Calder. Reducing delay with dynamic selection of compression formats. In *IEEE HPDC*, 2001.

[19] N. Motgi and A. Mukherjee. Network conscious text compression system (nctcsys). In *IEEE ICIT*, 2012.

[20] C. Pu and L. Singaravelu. Fine-grain adaptive compression in dynamically variable networks. In *IEEE ICDCS*, 2005.

[21] G. Ray, J. R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[22] M. A. Roth and S. J. V. Horn. Database compression. *SIGMOD Record*, 22(3), 1993.

[23] D. Salomon. *Data compression: The complete reference*. Springer, 2006.

[24] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *IEEE IPDPS*, 2009.

[25] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *PACT*, 2012.

[26] StreamBase Systems. http://www.streambase.com. retrieved May, 2011.

[27] S. Sucu and C. Krintz. ACE: A resource-aware adaptive compression environment. In *IEEE ITCC*, 2003.

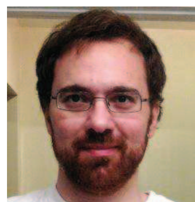
[28] D. Turaga, H. Andrade, B. Gedik, C. Venkatramani, O. Verscheure, J. D. Harris, J. Cox, W. Szewczyk, and P. Jones. Design principles for developing stream processing applications. *Software: Practice & Experience*, 40(12), 2010.

[29] Storm project. http://storm-project.net/. retrieved May, 2012.

[30] Y. Wiseman and K. Schwan. Efficient end-to-end data exchange using configurable compression. In *IEEE ICDCS*, 2004.

[31] R. Wolskia, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Springer FGCS*, 15(5-6), 1999.

[32] S4 distributed stream computing platform. http://www.s4.io/. retrieved May, 2012.



Buğra Gedik is with the Department of Computer Engineering, İhsan Doğramacı Bilkent University, Ankara, Turkey. Prior to that he was with the IBM T. J. Watson Reserach Center, NY, USA. He has obtained his Ph.D. degree in Computer Science from Georgia Institute of Technology, USA, and his B.S. degree in Computer Engineering and Information Science from Bilkent University, Turkey. Dr. Gedik's research interests are in distributed data-intensive systems with a particular focus on stream computing.