TerseCades: Efficient Data Compression in Stream Processing

Gennady Pekhimenko, *University of Toronto;* **Chuanxiong Guo,** *Bytedance Inc.;* **Myeongjae Jeon,** *Microsoft Research;* **Peng Huang,** *Johns Hopkins University;* **Lidong Zhou,** *Microsoft Research*

https://www.usenix.org/conference/atc18/presentation/pekhimenko

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

TerseCades: Efficient Data Compression in Stream Processing

Gennady Pekhimenko
University of Toronto

Chuanxiong Guo
Bytedance Inc.

Myeongjae Jeon
Microsoft Research

Peng Huang
Johns Hopkins University

Lidong Zhou
Microsoft Research

Abstract

This work is the first systematic investigation of stream processing with data compression: we have not only identified a set of factors that influence the benefits and overheads of compression, but have also demonstrated that compression can be effective for stream processing, both in the ability to process in larger windows and in throughput. This is done through a series of (i) optimizations on a stream engine itself to remove major sources of inefficiency, which leads to an order-of-magnitude improvement in throughput (ii) optimizations to reduce the cost of (de)compression, including hardware acceleration, and (iii) a new technique that allows direct execution on compressed data, that leads to a further 50% improvement in throughout. Our evaluation is performed on several real-world scenarios in cloud analytics and troubleshooting, with both microbenchmarks and production stream processing systems.

1 Introduction

Stream processing [7, 18, 22, 41, 1, 10, 11, 66, 48, 56, 57, 65, 21, 17] is gaining popularity for continuous near real-time monitoring and analytics. It typically involves continuous processing of huge streams of machine-generated, timestamped measurement data. Examples include latency measurements [35], performance counters, and sensor readings in a wide variety of scenarios such as cloud systems and Internet of Things (IoT) [2, 3]. In order to meet near real-time requirements, stream processing engines typically require that streaming data (coming in huge volumes) reside in the main memory to be processed, thereby putting enormous pressure on both the capacity and bandwidth of the servers' main memory systems. Having high memory bandwidth while preserving capacity is known to be difficult and costly in modern DRAM [44, 19, 63]. It is therefore important to explore ways, such as data compression, to relieve this memory pressure.

This paper presents the first systematic investigation of stream processing with data compression. The low-latency, mostly in-memory processing characteristics make data compression for stream processing distinctly different from traditional data compression. For example, in database or (archival) file systems, a sophisticated compression scheme with high compression ratio [68, 27, 37, 47] is often desirable because its overhead can be overshadowed by high disk latency. We start by observing opportunities for significant (orders of magnitude) volume reduction in production cloud measurement data streams and real-world IoT data streams, processed in real stream queries for cloud analytics and troubleshooting purposes, as well as for IoT scenarios. The high redundancy in the streaming data sets is primarily due to the synthetic and numerical nature of these data sets, including, but not limited to, timestamps, performance counters, sensor, and geolocation data. This key observation creates an opportunity to explore efficient encoding mechanisms to explore streaming data redundancy, including lossless and lossy compression (that is *harmless* with respect to specific queries output). For example, timestamps in the data streams are highly compressible even through simple lossless encoding mechanisms, such as variable-length coding [54] and base-delta encoding [53, 60]. By knowing the semantics of generic window-based streaming operators, we can further improve the benefits of compression by reducing the overprovisioning of the timestamps accuracy *without* affecting the produced results. The potential we have identified (for several representative streaming datasets) is likely to apply to other machine-generated time series as well.

Volume reduction, however, does not necessarily lead to proportional improvement in end-to-end throughput, even on a state-of-the-art stream engine such as Trill [21]. Our evaluation shows that an 8× reduction in data volume translates to less than 15% improvement in throughput on Trill, even without considering any encoding cost. This is because memory bandwidth is not yet

the bottleneck thanks to significant overhead elsewhere in the system. We therefore build TerseCades[1], a lean stream processing library that optimizes away the other major bottlenecks that we have identified in the existing stream engines, using techniques such as array reshaping [67], static memory allocation and pooling [42, 26], and hashing. TerseCades provides a vastly improved and competitive baseline (by an order of magnitude in throughput over Trill), while making a strong case for compression in streaming context.

Driven by real streaming queries on production data, we have identified factors that influence the benefits and overheads due to data compression, and proposed a series of optimizations to make compression effective for stream processing. This includes the use of SIMD instructions for data compression/decompression, hardware acceleration (using GPUs and FPGAs), as well as supporting execution directly on compressed data when it is feasible. To demonstrate the end-to-end benefits of our design, we have implemented compression support with the optimizations on TerseCades. Our evaluation shows that, altogether, these optimizations can improve the throughput by another 50% in TerseCades, on top of the order of magnitude improvement over Trill, while significantly improving processing capacity diverse temporal windows due to reduced memory footprint.

In summary, our contributions are as follows. (1) We identify major bottlenecks in a state-of-art stream engine and develop TerseCades that provides an order of magnitude higher throughput. (2) We characterize representative data streams and present compression algorithms for effective in-memory stream processing. (3) We implement these compression algorithms along with a set of optimizations (e.g., direct execution on compressed data) on TerseCades, improving throughput by another 50%.

2 Is Compression Useful for Streaming?

A natural starting point to assess the usefulness of compression for streaming is to check (i) whether data streams are compressible and (ii) whether data volume reduction from compression improves the throughput of stream processing. To do so, we perform a set of analysis using the Pingmesh data streams of network reachability measurements [35] from production data centers, with respect to motivating real data set for data-center network diagnosis. We then use Trill [21], a state of the art high-performance streaming library, and the STREAM [13] benchmark suite to evaluate the effect of data volume reduction on throughput.

[1] TerseCades = Terse (for compression) + Cascades (for streaming). Appropriately several characters in Cascades get 'compressed'.

2.1 Streaming Data Compressibility

For compressibility, we examine the Pingmesh data records. Major fields are listed in Table 1, and here we focus on two important fields: (i) 8-byte integer `timestamp` to represent the time when the request was issued, and (ii) 4-byte integer `rtt` values to represent request round-trip-time (in microseconds).

Stream processing operates on batches of data records that form *windows*. Our analysis on those batches reveals the potential of significant volume redundancy that can be easily exploited. For example, the `timestamp` values are often within a small range: more than 99% of the values in a 128-value batch differ in only 1 lower-order byte. This potentially allows efficient compression with simple lossless compression schemes such as Base-Delta encoding [53, 60] and variable-length coding [54] to achieve a compression ratio around 8× or more. Similarly, the `rtt` values for successful requests are usually relatively small: 97% values need only two bytes. This data can be compressed by at least a factor of 2.

While lossless compression can be effective in reducing data redundancy, we observe that in many real scenarios it is profitable to explore *lossy* compression without affecting the correctness of the query results. For example, in queries where timestamps are used in a windowing operator only for assigning a record to a time window, we can replace multiple timestamps belonging to the same window with just one value that maps them to a particular window. We provide more details on lossy compression in Section 3.3.

2.2 Compressibility ⇏ Performance Gain

We further study the effect of data volume reduction on stream processing using Trill, driven by a simple *Where* query that runs a filter operator on a single in-memory data field. We use two versions of the data field (8 and 1 bytes) to simulate an ideal no-overhead compression with a compression ratio of 8. This query performs minimum computation, does not incur any compression/decompression overhead, allowing Trill to focus on the actual query computation. Figure 1 shows the results.[2]

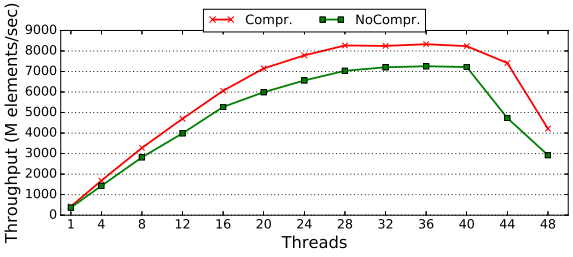

Figure 1: Throughput with data compression in Trill.

[2] Section 4 describes the methodology and system configurations.

As expected, the 1-byte compressed (*Comp*) version consistently outperforms the 8-byte non-compressed (*NoComp*) version. However, the amount of the improvement is relatively small (only 13-15%), compared to a factor of 8 reduction in memory traffic. This indicates that query processing in Trill is not memory-bound even when the query executes simple computation (e.g., a filter operator).

To understand the source of such inefficiency, we have run and profiled a diverse set of queries (including filter query and groupby query) using Trill. Our profiling shows that for the filter query (and similarly for other stateless queries), most of the execution time is spent in functions that generalize semantics in streaming data construction. In particular, for each incoming event, the filter query (1) performs just-in-time copy of payloads to create a streameable event (memory allocation) and (2) enables flexible column-oriented data batches (memory copying and reallocation). These operations account for more than two-thirds of the total query processing time, with limited time spent on the query operator itself. The second major overhead is inefficient bit-wise manipulation; 46% of the time is spent on identifying what bit should be set when the previous bottleneck is fully removed. For the groupby query, more than 90% of the time is spent on manipulating the hash table (e.g., key lookups), which holds the status of the identified groups. While adding concurrency mitigates such costs, they remain the largest.

In summary, we conclude that the state-of-the-art streaming engines such as Trill are not properly optimized to fully utilize the available memory bandwidth, limiting the benefit of reduced memory consumption through data compression. In our work, we will address this issue by integrating several simple optimizations that make streaming engines much more efficient and consequently memory sensitive (Section 3.1).

2.3 Compressibility ⇒ Performance Gain

To understand whether the limitations we observe with Trill are fundamental, we look at the performance of the STREAM [13] benchmark suite, which performs simple streaming operators such as copy, add, and triad on large arrays of data[3], without the overhead we observe in Trill.

Figure 2 shows the throughput of the *Add* benchmark for three different cases: (i) *Long* – 64-bit unsigned integer, (ii) *Char* – 8-bit char type (mimic 8× compression with *no* compression/decompression overhead, (iii) *CharCompr.* – compressing 64-bit values to 8-bit using Base-Delta encoding [53].

We draw two major conclusions from this figure. First, when the amount of data transferred is reduced 8× (going from Long to Char), the resulting throughput also increases proportionally, from the maximum of around 7 Billion elements/sec for *Long* all the way to the maximum of 52 Billion elements/sec with *Char*). The significant throughput improvement indicates that, due to the absence of other artificial bottlenecks (e.g., memory allocation/deallocation, inefficient bit-wise manipulation, and hashmap insertions/searches), the throughput of this simple streaming engine is limited only by the main memory bandwidth, and compression reduces the bandwidth pressure proportionally to the compression ratio. Second, using a realistic simple compression mechanism, e.g., *CharCompr.* with Base-Delta encoding, still provides a lot of benefits over uncompressed baseline (the maximum increase in throughput observed is 6.1×), making simple data compression algorithms an attractive approach to improve the performance of stream processing. At the same time, it is clear that even simple compression algorithms incur noticeable overhead that reduces the benefits of compression, hence the choice of compression algorithm is important.

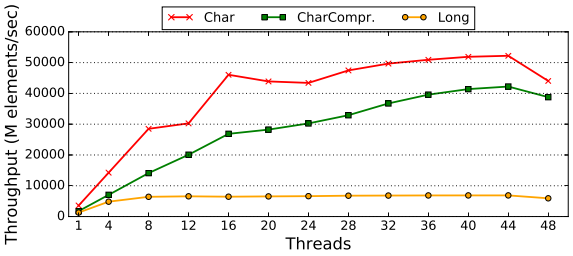

Figure 2: Add results.

3 Efficient Compression with TerseCades

In this section we first describe the key optimizations (which we refer to as first-order) that are needed for a general streaming engine to be efficient. We then describe the design of a single-node streaming system that supports generic data compression. Finally, we show the reasons behind the choice of compression algorithms we deploy in TerseCades, hardware-based strategies to minimize the (de)compression overhead (using SIMD, GPU and FPGA), as well as less intuitive (but very powerful) optimizations such as direct execution on compressed data.

3.1 Prerequisites for Efficient Streaming

Our initial experiments with Trill engine (§2) show that in order to make streaming engines more efficient, several major bottlenecks should be avoided. First, dynamic memory allocation/deallocation is costly in most operating systems, and permanent memory allocation for every

[3]For our purposes, we evaluate STREAM not only on float/double data, but also for different integer types.

window (or even batch within a window) in streaming engine significantly reduces the overall throughput. This happens because the standard implementation of streaming with any window operator would require dynamic memory allocation (to store a window of data). One possible strategy to address this problem is to identify how much memory is usually needed per window (this amount tends to be stable over time as windows are normally the same size), and then use *fixed memory allocation* strategy – most of the memory allocation happens once and then reused from the internal memory pool. In TerseCades we use profiling to identify how much memory is usually needed for a particular size window, allocate all this memory at the beginning, and then only allocate more memory if needed during the execution.

Second, implementation of certain streaming operators, e.g., *GroupApply*, requires frequent operation on hashmap data structures. Similarly, many common integral data types such as strings, might require a lot of memory if stored fully (e.g., 64 bytes for the server IDs), but can be efficiently hashed to reduce space requirements. Unfortunately, the standard C++ STL library does not provide this efficiency. To address this problem, we implement our own hashmap data structure with corresponding APIs taking into the account specifics of our streaming data.

Third, efficient implementation of filtering operators (e.g., *Where*) requires efficient bit-vector manipulation. For example, when running a simple *Where* query with a single comparison condition (e.g., `Where (errorCode == 0)`) with Trill streaming engine, we observe that about 46% of the total execution time is now related to simple bit-wise manipulation (1 line of the source code using standard C# data structures). Unfortunately, this huge overhead limits the benefits of any further performance optimizations. In our design, we implemented our own simple bit-wise representation (and the corresponding APIs) for filtering operators using C++ that significantly reduces the overhead of filtering. Altogether, these optimizations allows us to improve the performance our system more than 3× as we will show in Section 5.

3.2 System Overview

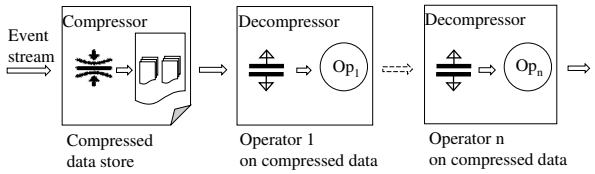

Figure 3: The streaming processing pipeline with compression and decompression.

Figure 3 shows our proposed TerseCades streaming processing pipeline in a single node. We will defer the discussion on how TerseCades is applied in the distributed system setting for monitoring and troubleshooting for a large cloud provider in Section 5.4, and in this section we focus on making this system efficient. We also note that single-node TerseCades system is generic, and both its design and optimizations behind it can be applied in other distributed streaming systems.

The major difference from traditional streaming processing, in Figure 3, is that external streaming events are first compressed before they are stored (typically in a column-oriented format that is usually more preferable for applications with high spatial locality). Note that the streaming operators also need to carry out decompression on all the compressed data before they access it (except for the cases where we use direct execution on compressed data described in Section 3.3).

The operators are chained together to form a pipeline. Different operators may work on different columns of the data, hence they may need to perform different decompression operations. Furthermore, some operators may need to produce new data sets from their input, and the newly generated data sets need to be compressed as well. This flow highlights the fact that compression/decompression operations are now on the critical path of the streaming engine execution, and have to be efficient to provide any benefits from tighter data representation .

3.3 Practical Compression for Streaming

One of the key contributions of this work is the efficient utilization of the existing memory resources (both bandwidth and capacity) by using simple yet efficient data compression algorithms. We observe that the dominant part of the data we use in stream processing is synthetic in nature, and hence it has a lot of redundancy (see Section 2 and 5) that can be exploited through data compression. In this section, we describe the key design choices and optimizations that allowed us to make data compression practical for modern streaming engines.

Lossless Compression. The key requirement of lossless compression is that the data after decompression should be exactly the same as before compression. The classical lossless compression algorithms include different flavors of Lempel-Ziv algorithm [68], and Huffman encoding [37, 27, 28, 47] and arithmetic coding [60, 36, 54, 4, 59]. These algorithms were proven to be efficient for disk/storage or virtual memory compression [62, 29, 9] and graphics workloads [60, 36, 54], but unfortunately most of these algorithms are too slow for compressing active data in memory. [4] As we will show in Section 5.1,

[4] Memory latencies are usually on the order of tens of nanoseconds [39]. Even when these algorithms were implemented as an ASIC design, e.g., IBM MXT design [8, 61], the overhead more than double the latency for main memory accesses.

software implementations of these algorithms are usually impractical.

To address this challenge, rather than using sophisticated dictionary-based algorithms, we decided to use simple arithmetic compression algorithm that was recently proposed in the area of computer architecture – Base-Delta encoding [53]. The primary benefits of this algorithm include its simplicity (e.g., only one addition is needed for decompression), and its competitive compression ratio for a wide range of data (e.g., rtts, timestamps, pixels, performance counters, geolocation data). Figure 4 shows how timestamp data can be compressed with Base-Delta encoding (8-byte base and 1-byte deltas).

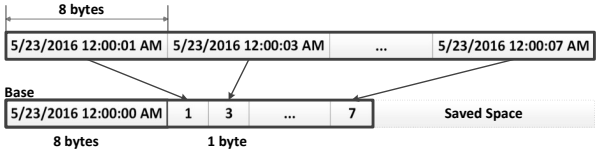

Figure 4: Base-Delta encoding applied to timestamps.

It turns out that this simple algorithm has several other benefits. First, it can be easily extended to a more aggressive lossy version that can still provide the output results that match lossless and uncompressed versions. Second, this algorithm is amenable to hardware acceleration using existing hardware accelerators such as GPUs and FPGAs, and using SIMD vector instructions available in commodity CPUs. Third, Base-Delta encoding preserves certain properties of the data (e.g., order) that can enable further optimizations such as direct execution on compressed data (Section 3.3).

Lossy Compression without Output Quality Loss. Lossy compression is a well-known approach to increase the benefits of compression at the cost of some precision loss. It is efficiently used in the areas where there is a good understanding of imprecision effect on the output quality, e.g., audio encoding, image compression. We observe that similar idea can be useful for some common data types in stream processing when the data usage and the effect of imprecision is also well understood.

For example, in troubleshooting scenario (§5.1), every record has a `timestamp` (8-byte integer) that is usually used only to check whether this timestamp belongs to a particular time window. As a result, storing the precise value of the timestamp is usually not needed and only some information to check whether the record belongs to a specific window is needed. Luckily, if the value already compressed with Base-Delta encoding, this information is usually already stored in the *base* value. Hence, we can avoid storing the *delta* values in most cases and get much higher compression ratio. For a batch of 128 `timestamp` values, the compression ratio can be as high as 128× for this data field, in contrast to about 8× compression ratio with lossless version. While this approach is not applicable in all cases, e.g., server IDs need to be precise, its impressive benefits while preserving the output quality made us consider using modified (lossy) version of Base-Delta encoding in our design.

Lossy Compression for Floating Point Data. The nature of floating point value representation makes it difficult to get high compression ratio from classical Base-Delta encoding. Moreover, getting high compression ratio with lossless compression algorithms on floating point data is generally more difficult [20, 14, 15]. Luckily, most of the scenarios using floating point values in streaming do not usually require perfect accuracy. For example, in several scenarios that we evaluated (§5), floating point values are used to represent performance counters, resource utilization percentage, geolocation coordinates, and sensor measurements (e.g., wind-speed or precipitation amount). In these cases, you usually do not need the precise values for all data fields to get the correct results, but certain level of precision is still needed.

We consider two major alternatives: (i) fixed point representation that essentially converts any floating point value into an integer value and (ii) using *lossy* floating point compression algorithms (e.g., ZFP [45]). The primary advantage of the first option is low overhead compression/decompression, because we can use Base-Delta encoding to compress the converted values. The primary benefit of the lossy floating point compression algorithms is that they usually provide higher accuracy than fixed-point representation. The lossy compression algorithm, called ZFP [45] that we use in our experiments, has the option to provide an accuracy bound for every value compressed. This option simplifies the usage of lossy compression since we only need to reason about data accuracy in simple terms (e.g., error bound per value is 10^{-6}). Moreover, this algorithm proved to have a very competitive throughput for both compression and decompression and allows to access the compressed data without decompressing it fully. Hence, in our design, we decided to use ZFP algorithm for floating point data.

Reducing the Compression/Decompression Cost. As we show in Section 2.3, even simple compression algorithms like Base-Delta encoding can add significant overhead. In this section, we will demonstrate how those overheads can be significantly reduced if we use the existing hardware to accelerate the major part of this overhead – data decompression.

Acceleration using SIMD instructions. Our original software implementation of Base-Delta encoding algorithm uses a simple add instruction to decompress a value based on its corresponding base and delta values. In streaming, usually many values are accessed at the same time, hence it is possible to reduce decompression overhead by using SIMD instructions, e.g., Intel AVX in-

structions (256-bit versions are available on most modern CPUs). By using SIMD instructions, we can reduce the overhead of decompression at least 4×, as four decompressions can be replaced with a single one. As we will show in Section 4.2, this optimization can significantly reduce the overhead of decompression that leads to the throughput close to the ideal compression case (with no compression/decompression overhead).

Hardware Acceleration: GPUs/FPGAs. Modern hardware accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) can be very efficient computational substrate to perform bulk compression/decompression operations [50, 31]. These accelerators are now available in commodity data centers for general-purpose use [55, 12, 16]. In our work, we also evaluate such a possibility by implementing Base-Delta encoding algorithm using CUDA 8.0 [49] on a GPU and using SystemVerilog on an FPGA. Our results (see Section 4.2) shows that by utilizing these accelerators, it is possible to perform the required data decompression (and potentially compression) without slowing down the main computation.

Direct Execution on Compressed Data. Many data compression algorithms require compressed data to be decompressed before it is used. However, performing each operation after data decompression can potentially lead to significant performance penalty. In streaming analytics, many operations are relatively simple and regular, allowing direct execution on the *compressed data itself*.[5] We find that we can run a set of stateless operators (e.g., Where) as well as aggregation operators (e.g., sum, min/-max, average, standard deviation, argmax/argmin, countif, distinct count, percentiles) on top of compressed data (assuming Base-Delta Encoding) more efficiently.

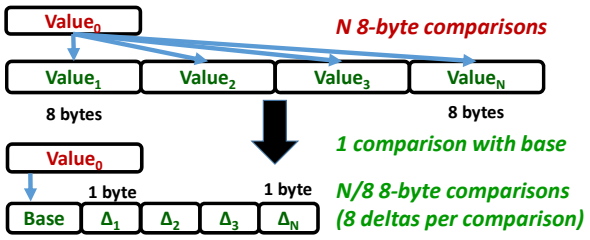

Figure 5: Direct execution on the data compressed with Base-Delta encoding.

Consider a simple *Where* query that performs a linear scan through an array on N values searching for a certain $value_0$ (see Figure 5). If this data is already compressed with Base-Delta encoding, then one simple strategy is to try to represent the searched value in the same base-delta format as batches of values in this array. If $value_0$ cannot be represented in this form (one comparison needed to test this), then this value is not in this batch, and there is no need to do any per-value comparisons. This avoids multiple (e.g., hundreds) comparisons per batch.

In cases where the search value can be represented similarly to the values in the batch, we still need to do value-by-value comparisons, but these values are now stored in a more narrow format (1-byte deltas in Figure 5). Instead of 8-byte comparisons, we can now group the deltas to do more efficient comparisons using SIMD instructions. This would reduce the number of comparisons by 8×. In summary, we can significantly improve the execution time for some common queries by utilizing the fact that data is stored in a compressed format to perform some operations more efficiently.

Generality of the Proposed Approach. There is a potential concern with the proposed approach due to limited generality and whether the benefits will be preserved when additional layers of indirection are added. Our current implementation supports a subset of queries from a LINQ-like proviver (e.g., *Where*, *Window*, *GroupBy*, *Aggregate* wih different operators inside of it, *Reduce*, *Process* and other operators are already supported), and is designed in a way, where it is possible to add support for new operators without significantly affecting the performance of existing ones. We currently leave it to the programmer to decide on whether they want their data being compressed using different proposed compression algo- rithms (similarly to how data compression is supported in Oracle databases). Our design favors column-oriented data allocation as it allows to get higher benefit from data locality in both DRAM and caches. It might not be al ways the best choice of each query, so we also leave the choice of memory allocation to the programmer (we pro- vide both row- and column-oriented options). Given these two inputs from the programmer, the framework then automatically allocates the memory in the form optimized for both streaming and compression, and performs execution on compressed data where applicable.

Our original intent was to perform our optimizations on top of existing state-of-the-art frameworks such as Trill, but as we show in Section 2, these frameworks are not properly tuned to exploit the full potential of existing memory subsystems. While we agree that their generality and ease of programming makes them a desirable choice in many situations, but we also envision TerseCades being further extended to be as general as these frameworks without sacrificing its performance.

4 Methodology and Microbenchmark

In this section, we will provide the detailed performance analysis of several key microbenchmarks and

[5] This idea has some similarity with the execution on encrypted data in homomorphic encryption [32], however in our case it is possible get performance even better than the uncompressed baseline.

demonstrate how different optimizations proposed in Section 3.3 affect their performance.

4.1 Methodology

In our experiments, we use two different CPU configurations. The first is a 24-core system based on Intel Xeon CPU E5-2673, 2.40GHz with SMT-enabled, and 128GB of memory, which is used in all microbenchmarks studies to have enough threads to put reasonable pressure on the memory subsystem. The second is a 4-core system based on Intel Xeon CPU E5-1620, 3.50GHz, SMT-enabled, and 16GB of memory. This system is used in all real applications experiments as it has better single-thread performance (especially higher per thread memory bandwidth). For our GPU experiments, we use NVIDIA GeForce GTX 1080 Ti with 11GB of GDDR5X memory. For FPGA prototyping, we use Altera Stratix V FPGA, 200MHz. In our evaluation, we use real applications scenarios (§5) and microbenchmarks from the STREAM suite [13]. We use *Throughput* (Millions of elements per second) and *Latency* (milliseconds) to evaluate streaming system performance; and *Compression Ratio* defined as uncompressed size divided by compressed size as the key metric for compression effectiveness.

4.2 Microbenchmark and Optimizations

SIMD-based Acceleration. To realize the full potential of data compression, it is critical to minimize the overhead due to data compression and especially decompression (that can be called multiple times on the same data). Luckily, the simplicity and inherent parallelism of the Base-Delta encoding algorithm allow to use SIMD vector instructions (e.g., Intel AVX) to perform multiple compressions/decompressions per instruction. Figure 6 shows the result of this optimization for *Add* benchmark from the STREAM benchmarks. We make two key observations from this figure.

First, when the number of threads is relatively small, this benchmark is more compute than memory limited. Hence reducing the computational overhead allows the *CharCompr.+V* version (compression plus vectorization) to almost completely match the ideal compression version (*Char*).[6] Second, when the number of threads increases (from 16 to 36), the additional overhead due to compression associated metadata becomes more important, and eventually when memory bandwidth becomes the only bottleneck, vectorization is not as useful in reducing the overhead anymore.

GPU/FPGA-based Acceleration. There are other hardware accelerators that can perform compression/decompression for Base-Delta encoding efficiently. For example, modern GPUs are suitable for massively parallel computations. We implemented the code that can perform multiple decompression operations in parallel using CUDA 8.0 [49] and tested this implementation using GeForce GTX 1080 Ti Graphics Card. Our results show that we can perform more than 32 Billion decompressions per second that is sufficient to satisfy the decompression rates required in realistic applications we will explore in Section 5. Note that this massive compute capability is frequently limited by the PCIe bandwidth that for our system was usually around 5-6 GB/sec.

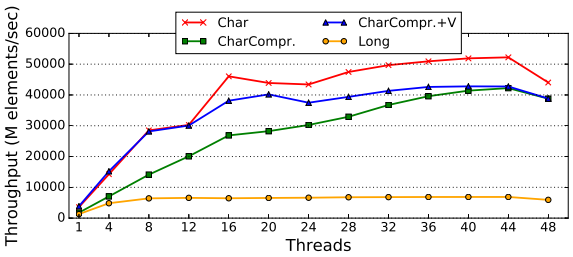

Figure 6: `Add` results with vectorization added.

Another option we explore is FPGA. We used SystemVerilog to implement the decompression logic and were able to run decompression at 200 MHz on a Stratix V FPGA board. We are able perform up to 744 Billion decompressions per second using this FPGA. Unfortunately, the bandwidth available through the PCIe again becomes the critical bottleneck limiting the number of decompressions we can perform. Nevertheless, it is clear that both GPUs and FPGAs can be efficiently used to hide some of the major data compression overheads.

Execution on Compressed Data. As we discussed in Section 3.3, the fact that the data is compressed usually comes with the burden of decompressing it, but it does not always have to be this way. There are several common scenarios when compressed data with Base-Delta encoding, can allow us to not only avoid decompression, but even execute the code faster. To demonstrate that, we take one benchmark called *Search* that essentially performs an array-wide search of a particular value (mimicking a very common *Where* operator in streaming). As we described in Section 3.3, when the data is represented in Base-Delta encoding, we take advantage of this fact and either completely avoid per value comparison within a batch (if the searched value is outside of the value range for this batch) or perform much more narrow 1-byte comparisons (8× less than in the original case).

Figure 7 presents the results of this experiment where *Compr.+Direct* is the mechanism that corresponds to compression with Base-Delta encoding and direct execution on compressed data as described in Section 3.3. Our key observation from this graph is that direct execution can not only dramatically boost the performance by

[6] Some additional overhead such as metadata and base storage overhead does not play a significant role here.

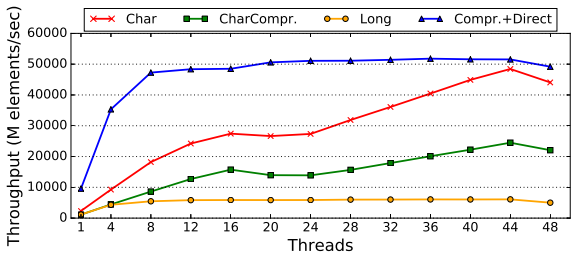

Figure 7: `Search` results with Direct Execution.

TimeStamp (8, **BD**)	ErrorCode (4, **EN+BD**)
SrcCluster (4, **HS+BD**)	DstCluster (4, **HS+BD**)
RoundTripTime(RTT) (4, **BD**)	

Table 1: Pingmesh record fields. Numbers in parenthesis are the bytes used to represent the field while letters are the compression algorithms we apply for that field. **BD**: base+delta; **HS**: string hashing; **EN**: enumeration.

avoiding the overhead of decompression (this is the performance gap between *Char*, ideal 8× compression with no overhead, and *CharCompr.*), but also significantly outperform the ideal compression case of *Char* (up to 4.1×). Moreover, it can reach almost the peak performance at just 8 threads, at which point it becomes fully memory bottlenecked, in contract to other cases where the peak performance is not reached until 44 threads are used. In summary, we conclude that direct execution on compressed data is a very powerful optimization that, when applicable, can by itself provide the relative performance benefits higher than that from data compression.

5 Applications

5.1 Monitoring and Troubleshooting

Pingmesh Data. Pingmesh [35] lets the servers ping each other to measure the latency and reachability of the data center network. Each measured record contains the following fields: timestamp, source IP address, source cluster ID, destination IP address, destination cluster ID, round trip time, and error code. Table 1 shows several of the fields that will be used in the queries in this paper. The measured records are then collected and stored. Data analysis is performed for dashboard reporting (e.g., the 50^{th} and 99^{th} latency percentiles), and anomaly detection and alerting (e.g., increased latency, increased packet drop rate, top-of-rack (ToR) switch down).

Pingmesh Queries. Here we describe implementation of several real queries on the Pingmesh data.

The query C2cProbeCount counts the number of error probes for the cluster-to-cluster pairs that take longer than certain threshold:

```
C2cProbeCount = Stream
  .HopWindow(windowSize, period)
  .Where(e => e.errorCode != 0 && e.rtt >= 100)
  .GroupApply((e.srcCluster, e.dstCluster))
  .Aggregate(c => c.Count())
```

The T2tProbeCount query is similar to the previous one, but uses Join to count the number of error probes for the ToR-to-ToR pairs:

```
T2tProbeCount = Stream
  .HopWindow(windowSize, period)
  .Where(e => e.errorCode != 0 && e.rtt >= 100)
  .Join(m, e => e.srcIp, m => m.ipAddr,
       (e,m)=> {e, srcTor=m.torId})
  .Join(m, e => e.dstIp, m => m.ipAddr,
       (e,m)=> {e, dstTor=m.torId})
  .GroupApply((srcTor, dstTor))
  .Aggregate(c => c.Count())
```

In the query, *m* is a table which maps server IP address to its ToR switch ID.

Compression Ratio, Throughput and Latency. In our experiments, we compare different designs that employ various compression strategies and optimizations: (i) *No Compression*, baseline system with all first-order optimizations described in Section 3.1, (ii) *Lossless* compression mechanism that employs Base-Delta encoding with simple mechanisms such as hashing and enumeration, (iii) *LosslessOptimized* mechanism that combines lossless compression described above with the SIMD accelation and direct execution on compressed data, (iv) *Lossy* compression mechanism that uses lossy version of Base-Delta encoding in the cases where precise values are not needed, (v) *LossyOptimized* mechanism that combines lossy compression with the two major optimizations described in (iii). In addition, we evaluate two other designs: *Trill* streaming engine as a backend and *NonOptimized* design where we use TerseCades without any of the proposed optimizations.

The average compression ratio for these designs is as follows: *Lossless** designs have an average compression ratio on 3.1×, *Lossy** designs – 5.3×, and all other designs have no compression benefits (as compression is not used). Figure 8 compares the throughput of all the designs. First, as expected, first-order optimizations are critical in getting most of the benefits of implementing more specialized streaming engine in C++, leading to performance improvement of 9.4× over *Trill* streaming engine. As we remove most of the redundant computational overheads from the critical path, the memory bandwidth becomes a new major bottleneck. The four designs with data compression support overcome this bottleneck that limits systems' throughput – 32.3 MElems/s (Millions of elements-records per second).

Second, both *Lossless* and *Lossy* compression can provide significant throughput benefits as they have high average compression ratios (3.1× and 5.3×, correspondingly). However the full potential of these mechanisms

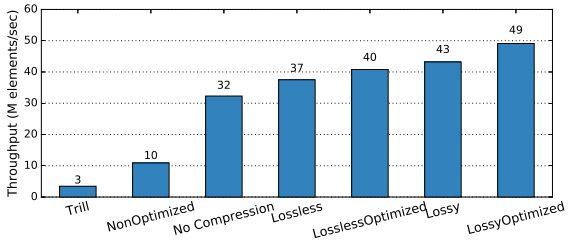

Figure 8: Throughput for the Pingmesh C2cProbeCount query. **Optimized** versions include both direct execution and SIMD optimizations.

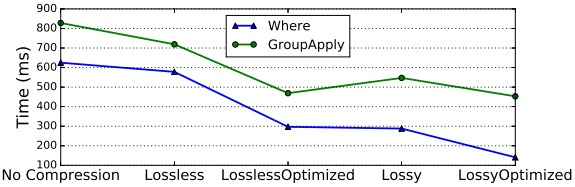

Figure 9: Execution times for Where and GroupApply operators used in the Pingmesh C2cProbeCount query. **Optimized** versions include direct execution and SIMD optimizations.

is only uncovered when they are used in conjunction with vectorization (that reduces the overhead of compression/decompression) and direct execution on compressed data that for certain common scenarios (such as *Where* operator) can dramatically reduce the required computation (over the baseline). We conclude that efficient data compression through simple yet efficient compression algorithms **and** with proper optimizations can lead to dramatic improvement in streaming query throughput (e.g., 14.2× for troubleshooting query we analyzed).

Figure 9 shows similar results on the performance (execution time) of two major operators, *Where* and *GroupApply* for first five designs. Two observations are in order. First, both operators significantly reduce their execution time due to efficient usage of data compression, and the highest benefit is coming again from the most aggressive design, *LossyOptimized*. Second, the benefits due to compression and corresponding optimizations are more significant for *Where* operator – 4.6× improvement between *NoCompression* and *LossyOptimized* designs. This happens because *Where* operator benefits dramatically from the possibility of executing directly on compressed data (reducing the number of comparisons instructions needed to perform this operator).

Join Operator: Throughput and Execution Time. In order to demonstrate the generality of our approach, we also evaluate another scenario (T2tProbeCount query) that uses the *Join* operator. Table 2 shows the throughput and the execution time of this scenario for two designs: *NoCompression* and *LossyOptimized*. In this scenario, the throughput benefits are still significant (almost 1.4×), but the reduction in the operator's execution time is relatively small (around 12%). This is because most of the benefits are coming from the reduction in bandwidth consumption that happens mostly outside of *Join* – in *HopWindow* operator, while most of the computation performed to implement the *Join* operator is coming from hash table lookups.

Mechanism	Throughput	Time
NoCompr.	27.7 MElems/s	2031 ms
LossyOptimized	38.3 MElems/s	1813 ms

Table 2: Throughput and Time (Join only) to perform the Pingmesh T2tProbeCount query that has Join operator.

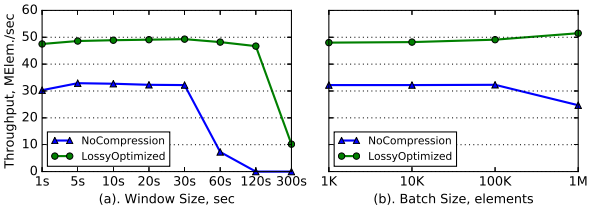

Figure 10: Throughput for the Pingmesh C2cProbeCount query with varying (a) window and (b) batch sizes.

Sensitivity to the Window Size. The window size used in the *HopWindow* operator can significantly affect the performance of the operators running after it. In order to understand if it is also a case for the troubleshooting scenario we consider, we study the performance of the two designs (*NoCompression* and *LossyOptimized*) on varying window size (from 1 second to 5 minutes).

Figure 10 (a) shows the results of this study. We make two observations. First, we observe that our proposed design, *LossyOptimized*, demonstrates stable throughput (around 50 MElems/sec) across different window sizes (from 1 to 120 seconds), with the variation below 5% in this range, and performance drops only at the 5-minute window. Second, in contrast to *LossyOptimized* design, the *NoCompression* design has a much shorter window of efficient operation (from 1 to 30 seconds).The primary reason for this is data compression that not only reduces the bandwidth consumed by the streaming engine, but also significantly reduces its memory footprint, allowing it to run on larger windows. Hence we conclude that compression allows to handle substantially larger windows (e.g., 4× larger) than the windows that can be efficiently handled without data compression.

Sensitivity to the Batch Size. In order to minimize the overhead of processing individual data elements, those elements are usually grouped by the streaming engines in batches. Hence the batch size becomes another knob to tune. We conduct an experiment where we vary the size

Mechanism	Throughput	Where	Group
NoCompr.	32.2 MElems/s	625 ms	828 ms
FrameRef.[33]	21.6 MElems/s	2453 ms	4078 ms
XPRESS [47]	8.5 MElems/s	7328 ms	13671 ms
LosslessOptimized	37.5 MElems/s	297 ms	469 ms
LossyOptimized	49.1 MElems/s	141 ms	453 ms

Table 3: Total throughput and time to perform Pingmesh query with optimized baseline without compression, two other compression algorithms, and our designs.

of the batch from 1K to 1M elements, and the results are shown in Figure 10 (b). We observe that in most cases the throughput of streaming engines are not very sensitive to the batch sizes in this range, except for one data point – the throughput of uncompressed design drop from 32.3 to 24.7 when going from 100K to 1M elements. Additional investigation shows that without compression for 1M elements the batch working set size exceeds the size of the available last level cache, limiting the benefits of temporal locality in the case of data reuse.

Sensitivity to the Compression Algorithms. In this work, we strongly argue that in order for compression algorithm to be applicable for streaming engine optimization, its complexity (compression/decompression overhead) should be extremely low and the algorithm itself has to be extensively optimized. We already showed that heavily optimized versions of arithmetic-based algorithm such as Base-Delta encoding can be efficient in providing significant performance benefits for streaming engines. We now compare our proposed designs with two well-known *lossless* compression algorithms used for in-memory data: FrameOfReference algorithm [33], arithmetic-based compression for low-dynamic range data, and XPRESS algorithm [47], dictionary-based algorithm that is based on LZ77 algorithm [68].

Our first comparison point is compression ratio, and as expected, both FrameOfReference and XPRESS outperform our *LosslessOptimized* algorithm in this aspect (compression ratios of 4.1× and 5.1×, respectively, vs. 3.1× for our design). However more importantly, as results in Table 3 indicate, both these algorithms prove not to be very practical for streaming engines, as their effect on throughput and execution time puts them below not only our proposed designs, but also significantly below uncompressed scenario. This happens because the cost of compression and decompression that are both on the critical path of execution outweighs the benefits of lower memory bandwidth consumption. We conclude that although it is important to have a compression algorithm with high compression ratio to provide reasonable performance improvements for streaming engines, it is even more critical to make sure those algorithms are efficient.

TimeStamp (8, **BD**)	Datacenter (3, **HS**)
Cluster (11, **HS**)	NodeId (10, **HS**)
VmId (36, **HS**)	CounterName (15, **EN**)
SampleCount (4, **BD**)	AverageValue (8, **ZFP**)
MinValue (8, **ZFP**)	MaxValue (8, **ZFP**)

Table 4: VM performance counter data fields. Numbers in parenthesis are the original sized of these fields, letters – compression algorithms used for them. **BD**: base+delta; **HS**: string hashing; **EN**: enum; **ZFP** [45].

5.2 IaaS VM Perf. Counter

Data. The cloud vendor regularly samples performance counters of an IaaS VM to determine a VM's health. If a VM is in an unhealthy state, recovery actions (e.g., relocate) will be taken to improve VM availability. Table 4 shows the fields for a performance counter record with the size to in-memory representations in the original analytics system. Each record contains the data source information (e.g., from which cluster, node and VM) and the actual values. At each regular interval, multiple such records of different types of counters will be emitted: e.g., CPU, network, disk. We use five datasets, i_1 to i_5, from different set of VMs in different timespan.

Queries. When processing these records, the data stream is grouped by timestamp and sources to get all the counters for a particular VM at each time point. We use a query to find the time and duration for a VM losing network activity: it first classifies the health of each perf counter group into different types (e.g., CPU active but network flat) and then scans the classified groups in ascending time for each unique VM to detect any type changes (e.g., from active to flat) and their durations.

Compressibility. Each performance counter record is represented with 111 bytes in memory in the original format, and a large portion of it can be compressed efficiently. For example, the `VmId` is a 36-character UUID string so that a VM can be universally uniquely identified across lifetime. But in the streaming scenario, in a given processing time window, the number of unique VMs tends not to be so large that they can be safely hashed to a 8-byte index. Note that absolute number of performance counters being emitted is large enough so that the hash table's overhead is amortized.

The compressibility can also come from batches of records rather than individual records. For example, the performance counter value is originally represented as a 8-byte double. The compressibility of a single record is not big (e.g., 2× if converted to integers). But efficient floating-point compression algorithm like ZFP can be applied across a stream of these counters to achieve high compression ratio. As Figure 11 shows, in certain runs, we can achieve near 6× compression ratio! The reason is that some VMs exhibit very regular performance patterns

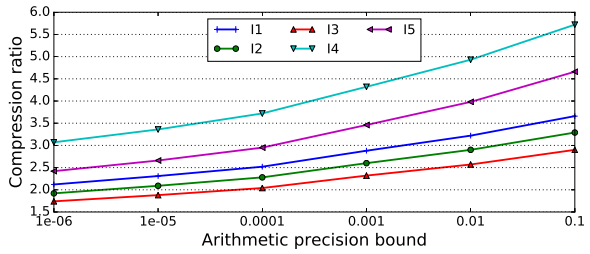

Figure 11: Compression ratio for various performance counters from VMs in a commercial cloud provider.

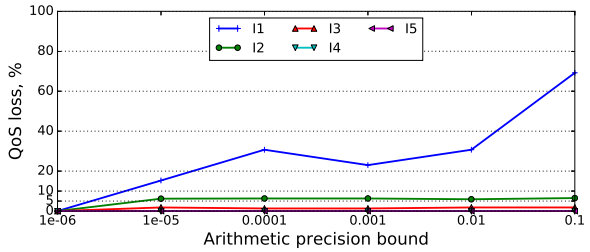

Figure 12: QoS metric loss for different compression ratios of performance counters.

(sometimes even constant) that can be exploited by ZFP. Of course, there are also some VMs that exhibit highly variable patterns, which will yield lower compression ratios (2.5× to 3× seen in our experiments).

Quality of Service. Algorithms like ZFP on floating-point values are lossy that can lose precisions when decompressed. Depending on the queries, the precision loss might sacrifice the QoS. This creates a trade-off between the compression ratio and the QoS level. We evaluated this trade-off using several queries on a set of real data coming from different regions' VMs in different timespan. The QoS loss metric is defined by the differences between the originally detected performance drop sessions with the new drop sessions: e.g., 1 additional session or missing session for a originally 100-session result is a 1% QoS loss. Figure 12 shows the result. We can see that for most datasets, the QoS loss level is low (below 5%), meaning that even aggressive lossy compression might be adopted without sacrificing QoS. For certain dataset, there is a high penalty (20% and more) because the absolute number of drop sessions are small.

5.3 IoT Data

Geolocation Data. This dataset contains GPS coordinates from the GeoLife project [2] by 182 users in a period of over three years. Figure 13 shows the average compression ratio of the dataset by using compression algorithms listed in Table 5. As we can see, these sensor data have significant redundancies because user movements tend not to have drastic changes. Even with an error bound of 10^{-6}, we can still achieve more than 4.5× compression ratio on average, creating significant opportunity for efficient real-time analytics over IoT data.

Latitude (8, **ZFP**)	Longitude (8, **ZFP**)
Altitude (4, **BD**)	TimeStamp (8, **BD**)

Table 5: Geolocation IoT data fields. Numbers and letters in parenthesis have the same meaning as Table 4.

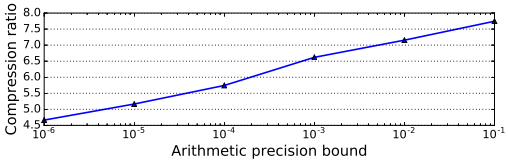

Figure 13: Compression ratio for GeoLocation IoT data.

Weather Data. We use another type of IoT data, 18,832,041 observations of weather data generated by various sensors during Hurricane Katrina in 2005 [3]. The measurements are stored as floating points in the data files but most of them are essentially integers due to the limited precisions of the sensors. Some metrics have a fixed number of digits after decimal points. They can be converted to integers as well to use integer compression algorithms like Base-Delta encoding. Across 18 metrics in the dataset, we can obtain an average of 3-4× compression ratios for each metric.

5.4 Real-World Implementation

We have built a distributed streaming system in our production data centers and implemented Pingmesh [35] usingTerseCades. In the original system, we used Trill [21] for streaming processing, which is now completely replaced with our new design. The whole system is composed of 16 servers each with two Xeon E5-2673 CPUs and 128G DRAM, running Windows Server 2016. Every server runs a frontend service and a backend service. The frontend services receive real-time Pingmesh data from a Virtual IP behind a load-balancer, and then partition the data based on the geo-region ID of the data and shuffle the data to the backend.

The whole system is designed to be fault tolerant – works well even if half services are down. The operations we perform are latency heatmap calculation at the 50^{th} and the 99^{th} percentiles, and several anomaly detections including ToR (Top of Rack) switch down detection. The aggregated Pingmesh input streaming is 2+ Gb/s, making it bandwidth-sensitive when properly optimized. The busiest server needs to process 0.5 millions events per second and uses 50% CPU cycles and 35GB memory. Using TerseCades to replace Trill, we reduce the sixteen servers to only one.

6 Related Work

Streaming System. Numerous streaming systems have been developed in both industry and literature [7, 18, 22,

41, 1, 10, 11, 66, 48, 56, 57, 65] to address the various needs for streaming processing: to name a few, Spark Streaming [66] applies stateless short-task batch computation to improve fault tolerance, MillWheel [10] supports user-defined directed graph of computation, Naiad [48] provides the timely dataflow abstraction to enable efficient coordination. One common requirement for these systems is to handle massive amount of unbounded data with redundancies. This motivates us to look into efficient data compression support in stream processing. Complimentary to these work, which focuses on high-level programming models in distributed environment, we focus on data compression in lower level core streaming engines that can potentially benefit these systems regardless of their high-level abstractions.

Memory Compression: Software and Hardware Approaches. Several mechanisms were proposed to perform memory compression in software (e.g., in the compiler [43] or the operating system [62]). While these techniques can be quite efficient in reducing applications' memory footprint, their major limitation is slow (usually software-based) decompression. This limits these mechanisms to compressing only "cold" pages (e.g., swap pages). Hardware-based data compression received some attention in the past [64, 8, 24, 30, 52]. However, proposed general-purpose designs had limited practical use either due to unacceptable compression/decompression latency or high design complexity, or because they require non-trivial changes to existing operating systems and memory architecture design.

Compression in Databases and Data Stores. Compression has been widely used to improve performance of databases [34, 58, 38, 25, 69, 5, 51, 46] and recent data stores [23, 9, 40] usually by trading-off overhead due to decompression for improved I/O performance and buffer hit rate. Some recent work investigates compression in the context of column-oriented databases [5, 6]that makes a few similar observations to our work: (i) adjacent entries in a column are often similar, (which helps improving compressibility), and (ii) some operators can run directly on compressed data to mitigate decompression costs (e.g., SUM aggregate on a run-length encoded column). The key difference in our work is that we apply compression in the streaming setting, and this puts significant limitations on the compression algorithm used (compared to offline data processing where latency is way less critical) that is now on the critical execution path. The proper choice of compression algorithm for streaming, reducing the key overheads of compression by using hardware acceleration, and using direct execution on compressed data (which not only avoids decompression, but actually executes faster than the baseline) are key contributions of our work that distinguish TerseCades from prior work on database compression.

One recent work, Succinct [9], supports queries that execute directly on compressed textual data (without indexes), significantly improving both memory efficiency and latency, but at the cost of complete redesign of how the data is stored in the memory. This is complementary to our work as our primary target is machine-generated, numerical data sets that proved to be more dominant in the streaming scenarios compared to textual data.

7 Conclusion

TerseCades is the first that attempts to answer the question of "Can data compression be effective in stream processing?". The design and optimizations of TerseCades answer these questions affirmatively. Our thorough studies and extensive evaluations using real stream workloads on production data further shed light on when and why compression might not be effective, as well as what can be done to make it effective. While our current implementation supports only a subset of operators supported by mature frameworks like Trill, we hope that by demonstrating the feasibility of data compression efficiency for streaming we will open the door for incorporating data compression in the next generation of stream processing engines.

References

[1] Apache Storm. http://storm.apache.org.

[2] GeoLife: Building social networks using human location history. https://www.microsoft.com/en-us/research/project/geolife.

[3] Linked sensor data. http://wiki.knoesis.org/index.php/LinkedSensorData.

[4] Aarnio, T., Brunelli, C., and Viitanen, T. Efficient floating-point texture decompression. In *System on Chip (SoC), 2010 International Symposium on* (2010).

[5] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (2006), SIGMOD '06, pp. 671–682.

[6] Abadi, D. J. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

[7] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. Aurora: A new model and architecture for data stream management. *VLDB J. 12*, 2 (2003), 120–139.

[8] Abali, B., Franke, H., Poff, D. E., Jr., R. A. S., Schulz, C. O., Herger, L. M., and Smith, T. B. Memory Expansion Technology (MXT): Software Support and Performance. *IBM J.R.D.* (2001).

[9] Agarwal, R., Khandelwal, A., and Stoica, I. Succinct: Enabling queries on compressed data. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), NSDI'15, pp. 337–350.

[10] Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. MillWheel: Fault-tolerant stream processing at Internet scale. *PVLDB 6*, 11 (2013), 1033–1044.

[11] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., ET AL. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1792–1803.

[12] Aws global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/, 2017. [Online; accessed 16-April-2017].

[13] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., NISHIZAWA, I., ROSENSTEIN, J., AND WIDOM, J. STREAM: The Stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (2003), p. 665.

[14] ARELAKIS, A., DAHLGREN, F., AND STENSTROM, P. HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48.

[15] ARELAKIS, A., AND STENSTROM, P. SC2: A Statistical Compression Cache Scheme. In *ISCA* (2014).

[16] Azure regions. https://azure.microsoft.com/en-us/regions/, 2017. [Online; accessed 16-April-2017].

[17] BAILIS, P., GAN, E., RONG, K., AND SURI, S. Prioritizing attention in fast data: Principles and promise. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data* (2017), SIGMOD '17.

[18] BARGA, R. S., GOLDSTEIN, J., ALI, M. H., AND HONG, M. Consistent streaming through time: A vision for event stream processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* (2007), pp. 363–374.

[19] BURGER, D., GOODMAN, J. R., AND KÄGI, A. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, USA, May 22-24, 1996* (1996), pp. 78–89.

[20] BURTSCHER, M., AND RATANAWORABHAN, P. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers 58*, 1 (Jan 2009), 18–31.

[21] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB 8*, 4 (2014), 401–412.

[22] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 668–668.

[23] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[24] CHEN, X., YANG, L., DICK, R., SHANG, L., AND LEKATSAS, H. C-pack: A high-performance microprocessor cache compression algorithm. *TVLSI* (2010).

[25] CHEN, Z., GEHRKE, J., AND KORN, F. Query optimization in compressed database systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (2001), SIGMOD '01, pp. 271–282.

[26] CURIAL, S., ZHAO, P., AMARAL, J., GAO, Y., CUI, S., SILVERA, R., AND ARCHAMBAULT, R. On-the-fly structure splitting for heap objects. *ISMM* (2008).

[27] DEUTSCH, P. Gzip file format specification version 4.3. https://tools.ietf.org/html/rfc1952, 1996.

[28] DEUTSCH, P., AND L. GAILLY, J. Deflate compressed data format specification version 1.3, rfc. https://www.ietf.org/rfc/rfc1951.txt, 1996.

[29] DOUGLIS, F. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Winter USENIX Conference* (1993).

[30] EKMAN, M., AND STENSTROM, P. A Robust Main-Memory Compression Scheme. In *ISCA* (2005).

[31] FOWERS, J., KIM, J.-Y., BURGER, D., AND HAUCK, S. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2015), FCCM '15, IEEE Computer Society, pp. 52–59.

[32] GENTRY, C., AND HALEVI, S. Implementing gentry's fully-homomorphic encryption scheme. Cryptology ePrint Archive, Report 2010/520, 2010. http://eprint.iacr.org/2010/520.

[33] GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering* (Feb 1998), pp. 370–379.

[34] GRAEFE, G., AND SHAPIRO, L. D. Data compression and database performance. In *In Proc. ACM/IEEE-CS Symp. On Applied Computing* (1991), pp. 22–27.

[35] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev. 45*, 4 (2015), 139–152.

[36] HASSELGREN, J., AND AKENINE-MÖLLER, T. Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2006), GH '06.

[37] HUFFMAN, D. A Method for the Construction of Minimum-Redundancy Codes. *IRE* (1952).

[38] IYER, B. R., AND WILHITE, D. Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), VLDB '94, pp. 695–704.

[39] JEDEC. DDR4 SDRAM Standard, 2012.

[40] KHANDELWAL, A., AGARWAL, R., AND STOICA, I. BlowFish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (2016), NSDI'16, pp. 485–500.

[41] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece* (2011).

[42] LATTNER, C., AND ADVE, V. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. *PLDI* (2005).

[43] LATTNER, C., AND ADVE, V. Transparent Pointer Compression for Linked Data Structures. In *Proceedings of the ACM Workshop on Memory System Performance (MSP'05)* (2005).

[44] LEE, D., GHOSE, S., PEKHIMENKO, G., KHAN, S., AND MUTLU, O. Simultaneous Multi Layer Access: A High Bandwidth and Low Cost 3D-Stacked Memory Interface. In *ACM TACO* (2015).

[45] LINDSTROM, P. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (Dec 2014), 2674–2683.

[46] MICROSOFT. Data compression. https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/, 2016.

[47] MICROSOFT. Ms-xca: Xpress compression algorithm. http://msdn.microsoft.com/enus/library/hh554002.aspx, 2016.

[48] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 439–455.

[49] NVIDIA. CUDA. https://developer.nvidia.com/cuda-toolkit, 2017.

[50] O'NEIL, M. A., AND BURTSCHER, M. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 7:1–7:7.

[51] ORACLE. Oracle advanced compression. http://www.oracle.com/technetwork/database/options/compression/overview/index.html, 2017.

[52] PEKHIMENKO, G., SESHADRI, V., KIM, Y., XIN, H., MUTLU, O., GIBBONS, P. B., KOZUCH, M. A., AND MOWRY, T. C. Linearly Compressed Pages: A Low Complexity, Low Latency Main Memory Compression Framework. In *MICRO* (2013).

[53] PEKHIMENKO, G., SESHADRI, V., MUTLU, O., GIBBONS, P. B., KOZUCH, M. A., AND MOWRY, T. C. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *PACT* (2012).

[54] POOL, J., LASTRA, A., AND SINGH, M. Lossless Compression of Variable-precision Floating-point Buffers on GPUs. In *Interactive 3D Graphics and Games* (2012).

[55] PUTNAM, A., CAULFIELD, A., CHUNG, E., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GRAY, J., HASELMAN, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., PETERSON, E., SMITH, A., THONG, J., XIAO, P. Y., BURGER, D., LARUS, J., GOPAL, G. P., AND POPE, S. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)* (June 2014), IEEE Press, pp. 13–24.

[56] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., AND ZHANG, Z. TimeStream: Reliable stream computation in the cloud. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), pp. 1–14.

[57] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., AND FREEDMAN, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 275–288.

[58] ROTH, M. A., AND VAN HORN, S. J. Database compression. *SIGMOD Rec. 22*, 3 (1993), 31–39.

[59] STRÖM, J., WENNERSTEN, P., RASMUSSON, J., HASSELGREN, J., MUNKBERG, J., CLARBERG, P., AND AKENINE-MÖLLER, T. Floating-point Buffer Compression in a Unified Codec Architecture. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), GH '08.

[60] SUN, W., LU, Y., WU, F., AND LI, S. DHTC: An Effective DXTC-based HDR Texture Compression Scheme. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), GH '08.

[61] TREMAINE, R. B., SMITH, T. B., WAZLOWSKI, M., HAR, D., MAK, K.-K., AND ARRAMREDDY, S. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro* (2001).

[62] WILSON, P. R., KAPLAN, S. F., AND SMARAGDAKIS, Y. The Case for Compressed Caching in Virtual Memory Systems. In *USENIX Annual Technical Conference* (1999).

[63] WOO, D. H., SEONG, N. H., LEWIS, D., AND LEE, H.-H. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *HPCA* (2010).

[64] YANG, J., ZHANG, Y., AND GUPTA, R. Frequent Value Compression in Data Caches. In *MICRO* (2000).

[65] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (2012), pp. 15–28.

[66] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 423–438.

[67] ZHAO, P., CUI, S., GAO, Y., SILVERA, R., AND AMARAL, J. Forma: A framework for safe automatic array reshaping. *TOPLAS* (2007).

[68] ZIV, J., AND LEMPEL, A. A Universal Algorithm for Sequential Data Compression. *IEEE TOIT* (1977).

[69] ZUKOWSKI, M., HEMAN, S., NES, N., AND BONCZ, P. Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering* (2006), ICDE '06, pp. 59–.