RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression

Francesco Fusco
ETH Zurich
Switzerland
fusco@tik.ee.ethz.ch

Michail Vlachos
IBM Research - Zurich
Switzerland
mvl@zurich.ibm.com

Xenofontas Dimitropoulos
ETH Zurich
Switzerland
fontas@tik.ee.ethz.ch

ABSTRACT

Network traffic archival solutions are fundamental for a number of emerging applications that require: a) efficient storage of high-speed streams of traffic records and b) support for interactive exploration of massive datasets. Compression is a fundamental building block for any traffic archival solution. However, present solutions are tied to general-purpose compressors, which do not exploit patterns of network traffic data and require to decompress a lot of redundant data for high selectivity queries. In this work we introduce RasterZIP, a novel domain-specific compressor designed for network traffic monitoring data. RasterZIP uses an optimized lossless encoding that exploits patterns of traffic data, like the fact that IP addresses tend to share a common prefix. RasterZIP also introduces a novel decompression scheme that accelerates highly selective queries targeting a small portion of the dataset. With our solution we can achieve high-speed on-the-fly compression of more than half a million traffic records per second. We compare RasterZIP with the fastest Lempel-Ziv-based compressor and show that our solution improves the state-of-the-art both in terms of compression ratios and query response times without introducing penalty in any other performance metric.

Categories and Subject Descriptors

D.4.1 [**Data**]: Coding and Information Theory— *Data compaction and compression*

General Terms

Algorithms

Keywords

Network monitoring, network traffic archives, data compression, NetFlow

1. INTRODUCTION

In many network monitoring tasks, it would be extremely useful to "travel back in time" and trace the causal nexus of events that led to important outcomes, like failures or security breaches. For this reason, an increasing number of applications need to efficiently search massive network traffic archives. For example, when a breach of credit card numbers is detected, analysts strive to find past traffic connections that are linked to the adversarial action to determine which systems and data were affected. Large-scale traffic repositories are also vital for network operators in validating claims against Service Level Agreements (SLAs), for scientists in studying Internet traffic, and for law enforcement agencies in lawful interception. All these applications need to support very efficient drill-down queries to find "needles in a haystack".

In the context of long term traffic archiving, compression is an essential enabling technology. It substantially reduces the size of an archive, while it directly relates with the efficiency of drill-down queries. Domain-optimized compression technologies have been researched, developed and sometimes standardized in several information technology fields, including, but not limited to bioinformatics [6, 21], biochemistry, and web information retrieval [26]. Previous experience shows that substantial gains can be obtained by designing compressors tailored to the data and requirements posed by specific domains.

In contrast, the de-facto approach to reduce the volume of network traffic repositories consists of applying general-purpose compression utilities, such as *gzip*, *bzip2*, and *lzop* [12, 17]. However, general-purpose compressors present limitations when used in the context of large-scale traffic repositories. The main shortcoming is that network traffic data archives exhibit distinct data patterns and pose specific requirements, which are not taken into account and exploited by general-purpose compressors.

In particular, the values stored in a traffic archive often share a common prefix (e.g., first few bits of IP addresses and timestamps), typically lie within a particular range (e.g., port numbers and protocols), and exhibit a large number of repetitions within a relatively short time window. A compressor optimized for network traffic data should exploit these patterns to *substantially reduce data volumes*.

Second, traffic data have to be compressed without introducing severe performance penalties when performing drill-down operations. Therefore, supporting efficient data retrieval from large-scale compressed repositories is necessary. Traffic archival solutions should provide mechanisms enabling *selective, fine-grained and high-speed* data decompression operations that minimize the amount of redundant data to be decompressed, especially in the case of drill-down operations targeting a small subset of the collected data. In contrast, general purpose compressors are optimized for compressing data in large blocks, which need to be retrieved, and fully decompressed, even for queries targeting a small part of the datasets.

Third, traffic archival solutions need to handle high rates of input streams, which are only expected to increase in the future. For example, present commercial flow collectors are designed to process up to hundreds of thousands flow records per second [1]. However, even higher rates can be observed caused by denial of service attacks, oversubscription of streams from many probes to a single collector or very high traffic rates on gateway routers of large institutions. Therefore, a compression algorithm designed for network traffic archival solutions has to provide *high-speed compression* of data streams.

Our work introduces 'RasterZip', a compressor optimized for the requirements and data patterns of network traffic archives. RasterZip compresses blocks of homogeneous network attribute types. It intelligently combines and adapts a number of compression techniques to exploit the *shared prefixes*, *bounded ranges*, and *frequent value repetitions* commonly observed in network traffic attributes. RasterZip first homogenizes each block based on a transposition that exploits the prefix structure and bounded range of traffic attributes and finally compresses it using an efficient online run-length encoding (RLE). Our encoding can sustain high-speed (de)compression exploiting features of modern CPUs instruction sets. More importantly, RasterZip introduces a novel decompression scheme based on **partial and adaptive decompression** to accelerate highly selective queries. With partial decompression, attribute values stored at specified positions can be retrieved from a RasterZip-compressed data block without requiring the entire block to be decompressed. Our adaptive decompression scheme dynamically selects at runtime between full or partial decompression based on a very light-weight decision classifier that predicts which strategy is expected to be the fastest.

We conduct an extensive evaluation of RasterZip using traffic traces from two different networks. The results show that: first, RasterZip can reach stream compression rates of more than half a million traffic flows per second, which is in par with the fastest general-purpose compressor (*lzop*). Second, decompression speed is better than *lzop*, which is one of the fastest decompressors. Third, RasterZip produces significantly more compact archives than *lzop.* Our experiments suggests an expected data reduction of approximately 22-24%. This directly suggests that compared to the state-of-the-art high-speed compressor, RasterZip reduces storage expenses for network traffic archives by an analogous amount.

In summary, our work makes the following contributions:

- **Domain-Specific Compressor.** RasterZip leverages network data patterns; enables interactive queries over archived data; supports online high-speed stream compression; and exploits features of modern hardware.

- **Novel Decompression.** RasterZip introduces a novel decompression scheme that provides fine-grained decompression granularity at the attribute level in order to accelerate queries for "needles in a haystack".

- **Outperform General-Purpose Compression.** Compared to the widely-used state-of-the-art *lzop* compressor, RasterZip accelerates queries, reduces expenses for storage by 22-24% due to more compact compression, and does not incur any penalty.

Although our design is focused on network traffic data, there are several monitoring domains and applications that demonstrate similar traits (shared prefixes, bounded data-ranges, and frequent data repetitions), and, therefore, can also benefit from our compressor. For example, data centers and cloud computing environments typically produce voluminous monitoring data to capture environmental (e.g. temperature) and performance metrics [12]. Such resource monitoring streams inherently contain numerous repetitions and shared prefixes and therefore can also benefit from RasterZip.

The remainder of the paper is structured as follows. In Section 2, we describe common strategies, recent trends and unsolved challeges in archiving network traffic flow records. Section 3 describes the internals of RasterZip. We evaluate our compressor in Section 5 and we discuss the cost of storage for large-scale traffic archives in Section 7. Finally, Section 8 concludes our paper.

2. BACKGROUND ON FLOW RECORD ARCHIVING AND SCOPE OF WORK

Modern network infrastructures come with built-in traffic monitoring functionalities. Switches and routers are equipped with embedded meters, referred to as *flow meters*, that analyze the network traffic to produce network flow records, which are then sent towards a centralized flow collector using an export protocol, such as Cisco's NetFlow [7]. A network flow is typically defined as the set of packets that have five header fields in common: source and destination IP addresses, source and destination port numbers, and the layer-4 protocol. Flow records carry detailed statistics describing each observed network flow, such as transferred bytes, number of packets, TCP flags, and timestamps.

Flow record analysis has many applications, including billing [10], traffic engineering [11], and anomaly detection [20]. Large-scale historical network flow record repositories are also used for forensics operations, for intrusion detection, and for conducting long-term network traffic studies. Such applications require flow storage infrastructures capable of storing high-speed data streams in a compressed format, while enabling highly efficient drill-down operations.

Existing Relational Database Management Systems (RDBMSs) present severe performance degradations when dealing with multi-gigabyte datasets [18] and offer limited support for data compression. A more commonly used strategy for collecting flow records consists of storing flow records or even *raw* NetFlow packets on disk as multiple files. This approach provides significantly higher insertion rates than RDBMSs, and allows users to compress their repositories using off-the-shelf general-purpose compression utilities. In this way, however, accessing the data involves time consuming decompression operations. NfDump [17], a widely-used flow processing tool, trades compression ratios for decompression speed by compressing data with the general-purpose compression utility *lzop* (based on the LZO algorithm), which is up to five times faster in decompression than *gzip*, but provides lower compression ratios [25].

Recent research has shown that advances in database technologies, stream compression and stream indexing, enable the development of more advanced flow record storage infrastructures that are more effective in compressing data and that are more suitable for drill-down operations.

- *Columnar databases* have been identified as a significantly better alternative to relational databases in the context of flow record collection [9, 14, 16]. By storing distinct attributes of flow records as separate physical columns, columnar databases offer better opportunities for compression [2] and are suited for queries that are highly selective at the attribute level (e.g., for queries that require just a few fields of a flow record to be accessed). These queries are common for after-the-fact analysis, where network operators are rarely interested in accessing the entire set of flow record attributes.

- *Compressed bitmap indexes* have been shown to be effective for indexing network traffic attributes [9, 13, 3]. They are a companion technology of columnar databases. Given a columnar database with k rows, a bitmap index is a binary array of length k that indicates the row positions, where an attribute assumes a specific value, e.g., `srcPort=`443. Bitmap indexes are intelligently compressed to allow for performing boolean operations in the compressed space between multiple compressed bitmap indexes. This enables to efficiently support multi-attribute queries, e.g., `srcPort=`X `AND srcIP=`Y, by combining the compressed bitmap indexes corresponding to columns of distinct attributes. Given a single- or multi-attribute query, compressed bitmap indexing enables to find very fast the n ($n \leq k$) row positions j_1 .. j_n that match the query. Compared to tree-based indexes (e.g., B-Trees), compressed bitmap indexes: i) are more compact, ii) provide high-speed indexing and iii) are optimized for read-only data. If data columns are compressed in small data blocks, the matching row positions can be exploited to selectively decompress the data blocks that store the requested data. This capability is very desirable for accelerating drill-down queries targeting a small portion of the dataset.

- *Stream-based reordering techniques* have been proposed in the context of network flow record collection to introduce better opportunities for compression [14]. The techniques profit from the fact that it is not necessary to strictly order flow records by time. Therefore, the order of incoming flow records can be altered to improve their compression.

In our previous work [14], we introduced a flow-record reordering technique, called online Locality Sensitive Hashing (**oLSH**), which is optimized for columnar databases. oLSH treats each multi-attribute flow-record as a multi-dimensional numerical vector and uses Locality Sensitive Hash functions [8] to bring flow records that are close to each other according to their euclidean distance to nearby stream positions.

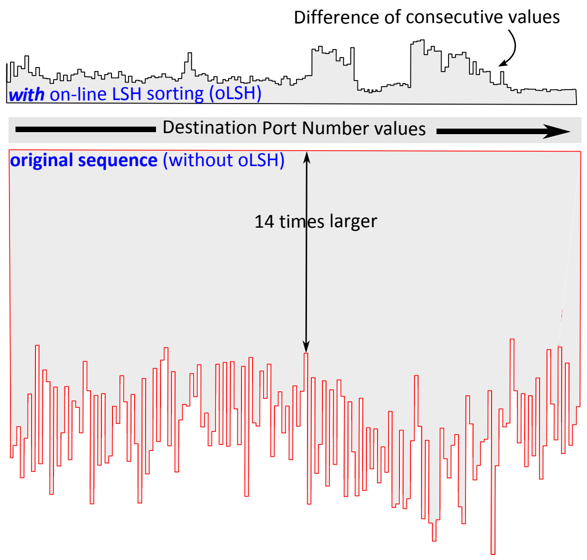


Figure 1: Effect of record reordering on column data homogenity. Storing the streaming records without reordering (bottom) results in a difference of consecutive values that is up to 14 times larger than when the online-LSH approach is used (top).

.

Figure 1 illustrates an example of the benefits of oLSH, which is described in detail in [14]. For one hour of network traffic we create a columnar archive for a multi-attribute flow record stream with and without the oLSH reordering enabled. For the destination port attribute, we compute the sum of differences between consecutive values for fixed-size blocks. The unordered stream shown on the bottom depicts significantly lower homogeneity: the cumulative difference between consecutive values is up to 14 times larger compared to the reordered stream using oLSH on top. It is evident that the reordering process is very effective in clustering values that belong to *adjacent ranges* and, thus, that share *common prefixes*.

In our previous work [14], we have also shown that by applying this transformation to a flow-record stream, the performance of the speed-optimized compressor LZO used to compress individual attributes separately in a columnar manner is substantially improved and in fact, LZO can match *gzip* in terms of compression ratio, while providing substantially higher decompression speed.

The main goal and outcome of our research (illustrated in Figure 2) is to move beyond general-purpose compressors by designing a domain-optimized compression algorithm that offers better compression ratios than *gzip* and *bzip2* while at least matching LZO, which was used in our previous work [14], both in terms of compression and decompression speed in the context of flow record archival.

Compared to general-purpose compressors, our compressor can fully profit from recent advances in flow record collection research including columnar databases, compressed bitmap indexes and stream-based flow record sorting techniques:

1. It is designed to exploit specific data patterns commonly present in network traffic attributes.

2. It is optimized for compressing data columns of colum-

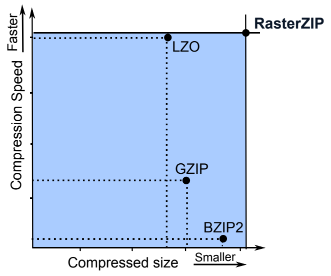


Figure 2: RasterZip versus general-purpose compressors.

nar databases in small data blocks allowing fine-grained decompression operations.

3. It can exploit bitmap indexes to retrieve attribute values stored at specific positions from a compressed block without performing a full decompression.

3. RASTERZIP COMPRESSION

RasterZip is a block-based compressor optimized for small data blocks intended to be used in columnar-databases. This means that the compression routine operates on blocks of *homogeneous* attribute values (e.g., timestamps). Our compressor has been designed to leverage the common prefix structure that is present in many network attributes (e.g., timestamps) and also produced by the oLSH transformation. RasterZip takes as input a column, e.g., of IP addresses, and *scans its bytes column-wise* (as shown in Figure 3) to take advantage of the observation that in this manner adjacent bytes are often identical. The column-wise order forms the input to our compressor, which is based on *run-length encoding* (RLE) principles. We call the new compression algorithm *RasterZip*, because we scan data in a raster-like fashion. In this section, we describe our compression techniques in detail.

3.1 Overview

The compression routine receives as input data blocks of homogeneous attribute values. Each block consists of m values of size n bytes. In our setting n corresponds, for example, to two bytes for port numbers, or four bytes for IPv4 addresses. Each data block of every attribute is treated as an $m \times n$ matrix of bytes, stored in row-major order.

Traditional data archiving approaches process the data row by row for reasons of efficiency and simplicity. Because of the data reordering, consecutive row-records share similar/adjacent values and hence have common prefixes. RasterZip compresses data efficiently by processing them column by column, where each cell represents one byte of the currently processed data attribute. This conceptual and practical distinction of our approach is shown in Figure 3. Note, that since rows share common prefixes, we expect very high compression for the first byte-columns. For the last byte-columns where higher variability is generally observed, lower compression ratios are to be expected.

Based on the above discussion, compression consists of two logical steps. First, the matrix is logically transposed and then the resulting matrix is traversed row-by-row to compress long sequences of repeated symbols. We provide an in-depth description of the two steps and the resulting encoding.

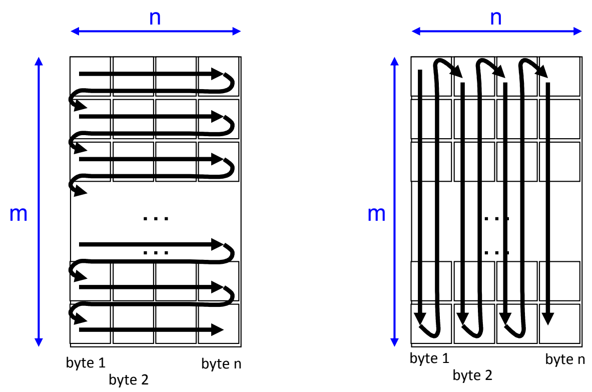


Figure 3: Left: Traditional compression approaches process data bytes in a row-wise fashion. Right: RasterZip scans the data bytes of an attribute in a columnar manner, in order to exploit the prefix structure of the input data.

1. **Transposition.** The $m \times n$ matrix is transposed, i.e., the i^{th} row becomes the i^{th} column for all $i \in \{1, \ldots, m\}$. Given that the first bytes of subsequent data items (i.e., subsequent entries in each of the first few columns in the original matrix) are often the same, transposing the matrix creates rows with many identical bytes in a row. As a small example, given an input of three IPv4 addresses (3×4 byte matrix), stored in memory as

```
10.4.20.22|10.4.20.23|10.4.21.24
```

the transposition yields the following representation:

```
10.10.10.4.4.4.20.20.21.22.23.24
```

This transformation results in long sequences of identical bytes not only for IP addresses, but also for other fields such as timestamps, flow duration, and so on.

We emphasize, that the above *transposition* is merely logical. One does not need to carry out the matrix transposition, but only traverse the data in a different order.

2. **Repeated symbol compression**: RasterZip efficiently compresses long sequences of repeated symbols (runs) using a technique based on run-length encoding (RLE) principles. RLE encodes the input stream as a list of pairs (r, l) where r is the symbol and l is the length. For example, RLE encodes the sequence 9,9,9,4,4,4,3 as (9,3),(4,3),(3,1).

RasterZip encodes runs in a more structured and efficient way that provides higher decompression performance compared to the standard RLE encoding, *particularly when highly compressible and incompressible content are interleaved in the input stream.* In addition, our encoding has been designed to enable random access on the compressed data. This is particularly useful for supporting partial data decompression.

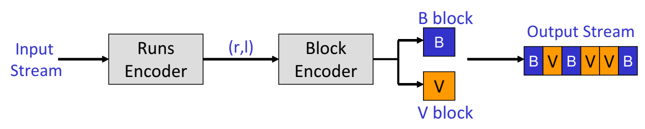


Figure 4: Flowgraph of RasterZip compression. Sequences of repeated symbols in the input stream are detected by the Runs Encoder and encoded as Bitmap (B) or Verbatim (V) blocks by the Block Encoder.

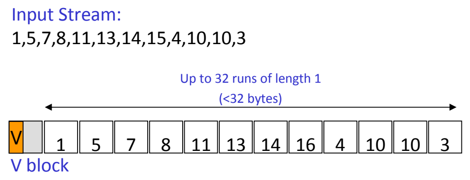


Figure 5: V-blocks are used to store content that is incompressible with RLE. V blocks store value repetitions of length 1. Runs of length 2 are split in two runs of length 1.

Figure 4 depicts a flowgraph of the compressor. The input stream for the compressor is the matrix given in column-major order. This is the currently processed data window, representing the output of the online-LSH component. The *Runs Encoder* scans the input stream and outputs to the *Block Encoder* pairs of the form (r, l) representing sequences of repeated symbols. The *Block Encoder* groups together runs in multiples of 32 and encodes each group by dynamically selecting between two different subblock types: a `Verbatim (V) block` and a `Bitmap (B) block`, which are described in the next section.

3.2 The Encoding

The encoding includes two different block types: `Verbatim (V) blocks` store RLE-incompressible content whereas `Bitmap (B) blocks` store RLE-compressible content.[1] Specifically:

1. **V-blocks** store a group of up to 32 runs whose length is exactly one. Runs of length 2 are stored as two runs of length 1 (i.e., the block encoder never receives runs of length 2 from the runs encoder). Therefore, such a block stores data that is hard to compress using RLE. A one-byte header is prepended. The header specifies the block type using 1 bit (i.e. that it is a `V-block`) and the number of runs which are stored in the block using five bits ($2^5 = 32$ runs). Because this block-type only stores runs of length 1, lengths need not be explicitly stored. Figure 5 depicts how an incompressible input stream is encoded with a `V-block`.

2. **B-blocks** encode up to 32 runs where at least one run has length greater than 1. `B-blocks` consist of:

   - A *header* (1 byte) specifying the block type and the number of runs (similarly to `V-blocks`).
   - A *presence bitmap* (32 bits) indicating which runs have length greater than 1.
   - A *Runs part* storing up to 32 runs.
   - A *Lengths part* storing up to 32 lengths.

   Therefore, unlike plain RLE, RasterZip does not store runs and lengths as pairs. Instead, runs are stored sequentially, followed by the sequence of the corresponding lengths. The reason for this becomes apparent in the following, where we describe how the `V-` and `B-blocks` are created in an online fashion.

   Note that a `B-block` stores mixed content of compressible and incompressible content. In fact, the presence bitmap (32 bits) exactly indicates which of the runs have length greater than 1. Recall that runs of length 2 are always treated as 2 consecutive runs of length 1. So, length is only explicitly stored for runs with length greater than 2. This is indicated by setting the corresponding bit in the presence bitmap to one.

   An example of a `B-block` is given in Figure 6. For the initial sequences of 10's and 9's the lengths are explicitly stored (4 and 3, respectively), whereas for the singular values that follow, no explicit length is recorded. This is captured in the presence bitmap, by setting the appropriate bits to 1 or 0.

   We also emphasize that, having allocated one byte for each length, we could have represented sequences of up to $2^8 = 256$ consecutive identical symbols. However, by not storing explicitly lengths 1 and 2, we can increase the maximum sequence length that can be represented to 258. In fact, since runs of length 1 are just marked in the bitmap and runs of length 2 are always treated as two runs of length 1, each length-byte in the `B-block` can store lengths in the range $[3, 258]$. This allows us to pack long sequences of the same symbol even more effectively.

[1] A block of attribute values is encoded into multiple subblocks of type `V` or `B`. The terms `V-block` and `B-block` refer to subblocks. The proper interpretation of the term block should be clear from the context.

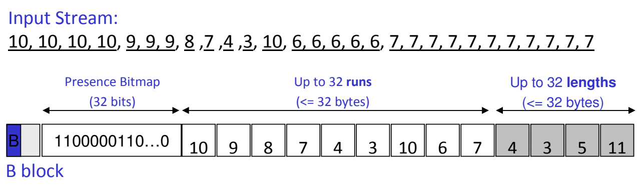


Figure 6: B-blocks store RLE-compressible content. B blocks encode up to 32 consecutive runs where at least one run has a length greater than 1. Only lengths greater than 1 are explicitly stored.

Online Blocks Creation Now we describe how the two types of blocks can be created by the Block Encoder in a streaming fashion. First, note that a `V-block` can be treated as a special case of a `B-block`, as it is a `B-block` whose presence bitmap does not have any bits set to 1. Therefore, the block creation always commences by assuming that the current block is a `B-block`. Whenever a run of length greater than 2 is observed, the presence bitmap is updated and the corresponding length is appended as well. After 32 runs are processed, we check whether the completed block contains at least one run of length greater than 2 by examining if at least one bit is set in the presence bitmap. In that case, the block is finalized and a new `B-block` is created. Otherwise, the presence bitmap is dropped and the block is changed into a `V-block` by simply flipping the first bit in the header byte. Pseudocode of this process is given in Algorithm 1.

Algorithm 1 Compress(input)

```
Input: Uncompressed $input$ stream
Output: Compressed $output$ stream
 1: {split runs of length two}
 2: while $(r, l) := getNextRun(input)$ do
 3:   {Update the bitmap}
 4:   if $l > 1$ then
 5:     $set\_bit(bitmap, curr\_run)$
 6:     $lengths[curr\_len{+}{+}] := l$
 7:   end if
 8:   {Add the current run}
 9:   $runs[curr\_run{+}{+}] := r$
10:   if $curr\_run = 32$ then
11:     {Output a V or a B block}
12:     if $bitmap = 0$ then
13:       $h := create\_header(V, curr\_run)$
14:       $output.appendHeader(h)$
15:       $output.appendBytes(runs, curr\_run)$
16:     else
17:       $h := create\_header(B, curr\_run)$
18:       $output.appendHeader(h)$
19:       $output.appendBitmap(bitmap)$
20:       $output.appendBytes(runs, curr\_run)$
21:       $output.appendBytes(lengths, curr\_len)$
22:     end if
23:     $curr\_run := 0$
24:     $curr\_len := 0$
25:     $bitmap := 0$
26:   end if
27: end while
```

3.3 Encoding properties and advantages

The proposed encoding presents several desirable properties with an emphasis on exploiting features of modern CPU architectures for enhanced performance. In particular RasterZip provides:

■ **Low memory footprint during compression**: Through the whole compression process, the algorithm is only required to allocate memory for a single RasterZip sub-block. A sub-block is very small: at most 69 bytes (header + bitmap + runs + lengths = 1 + 4 + 32 + 32). We also note that the average sub-block size is even smaller in practice. We observed an average of 40 bytes in our experiments. The cache-line size on modern processors is 64 bytes; therefore, at most 2 cache-lines are used during the compression. This leads to a low number of cache-misses and enhanced performance of the proposed compression algorithm. In contrast, popular compression algorithms, such as the Lempel-Ziv based compression schemes, use hash tables or dictionaries to store previously seen data [23], and as such, result in many cache misses in practice.

■ **Efficient handling of incompressible content**: RasterZip encodes data that is hard to compress using RLE (i.e., the length of each run is less than 3) with a `V-block`. This block-type enables fast decompression of incompressible data, because decompressing a `V-block` merely requires copying up to 32 bytes to the output, which can be implemented using the *memcpy()* function. On the contrary, decompressing hard to compress data encoded with the original RLE encoding requires a *conditional branch* (if statement) for every symbol, even if the length is as short as 1 or 2.

The next two points show how RasterZip exploits features offered by modern CPU architectures. These characteristics are useful primarily during the decompression process, as they allow the efficient traversal of RLE subblocks and the computation of the *span* of each subblock.

■ **Fast traversing of compressed data**: The simple structure of `V` and `B` blocks allows compressed data to be traversed quickly. A `V-block` storing n runs always consists of $1 + n$ bytes, and n can be read from the first byte after masking the three most significant bits to 0. Similarly, the size of a `B-block` is $1 + 4 + n + b$ bytes, where b corresponds to the number of bits set to one in the bitmap. The value of b can be efficiently computed using specialized *bit counting* instructions offered on modern processors. For example, Intel introduced POPCNT to the SSE4.2 instruction sets, which offers a dedicated assembly instruction to perform the bit counting operation.[2]

■ **Efficient computation of the block span**: given a block, its *span* is the number of bytes of uncompressed data stored in it. For both block types, the span can be efficiently computed: the span of a `V-block` is the number of runs that it stores. Computing the span of a `B-block` requires summing up the lengths greater than 2 and adding the number of runs with length 1. More formally:

$$Span_B = (n - b) + \sum_{i=1}^{b} l[i]$$

where n is the number of runs, b denotes the number of bits set to 1 in the bitmap, and $l[i]$ is the length of the i^{th} run whose length exceeds 2. Since b is at most 32, and therefore the array of lengths cannot be longer than 32, it is possible to *unroll the loop* for this computation using a switch statement with 32 entries. This optimization enables the fast computation of the block span, which is an essential component of the decompression process.

Additional properties of the encoding scheme: The described encoding represents runs in batches of 32. However, the number of runs encoded in `V-` or `B-blocks` is in principle a tunable parameter[3]. In practice, there are trade-offs to be considered: i) a smaller number of runs corresponds to finer-grained decompression granularity and lower performance when copying individual `V-blocks` to the output, and, ii) a larger number of runs favors `B-` over `V-blocks`, making the compressor less capable of mixing RLE-compressible and incompressible content efficiently. We fix the number of runs to 32, which resulted to be a good compromise between compression ratio and decompression performance.

[2]Modern compilers, such as GCC, offer the intrinsic function *__builtin_bitcount(unsigned int)* for using POPCNT in C. The *-msse4* compilation flag instructs the GNU GCC compiler to use the POPCNT instruction.

[3]The 1-byte header of both `V-` and `B-blocks` can be accommodated for up to 128 runs by using 7 instead of 5 bits to store the number of runs belonging to the block ($2^7 = 128$ runs). In the case of `B-blocks` the maximum number of runs has to be the same as the length, expressed in bits, of the presence bitmap.

4. DECOMPRESSION

Recall that for a certain attribute, a columnar archive contains multiple RasterZip-compressed blocks, which in turn are made of multiple RLE-subblocks (`V` or `B`). At runtime, user queries are evaluated using an index (e.g., a compressed bitmap index), which returns the row positions that match within the data column. Therefore, the query processor receives as input a list of row positions from which it determines the blocks that need to be decompressed.

The row positions list also specifies the positions of the attribute values stored by a specific RasterZip-compressed block that belongs to the result set. Compared to general-purpose compressors, our approach can exploit this information to provide partial data decompression: it decompresses only the RLE subblocks that are part of the result set.

We first start by examining two block decompression strategies: a *full decompression* and a *partial decompression*. The latter approach decompresses only the RLE-subblocks that contain parts of the result set. Intuitively the first approach is better when a block contains many results; the second approach when only few results need to be retrieved. Finally, we show how to adaptively select between full and partial decompression at runtime. We introduce an adaptive decompression component that intelligently shifts between full- or partial decompression, according to which approach is expected to result in lower decompression time.

4.1 Full-Decompression (FD)

Assume that a compressed block is marked for decompression because it contains results for a given query. The block contains m data items each with size of n bytes and the index shows the row positions $r_1, \ldots, r_k$ $(k \leq m)$ within the block. The Full-Decompression strategy is shown in Figure 7. It decompresses the entire block (i.e., all RLE-subblocks) which is then stored in column-major order. Block transposition back to row-major order does not need to be performed, because the required data rows can be retrieved directly, as we explain below.

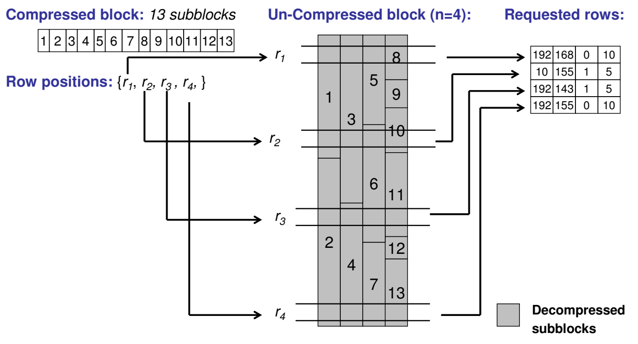


Figure 7: Full-Decompression (FD) decompresses the entire set of RLE subblocks and then uses the row positions to compute the indices of the bytes that need to be retrieved in the uncompressed matrix stored in column-major order.

Consider each decompressed archive block as an $m \times n$ matrix. Each required row position r can be retrieved by accessing the bytes at the offsets $r, r+m, .., r+(n-1)m$, since the offset between the bytes that belong to a certain row is exactly m.

An example of the retrieval procedure is given in Figure 8, which on the left illustrates a block with three IPv4 addresses and 4 bytes per row. Because the decompressed matrix is in column-major order, as shown in the right of Figure 8, to access a row r at position x, the decompressor traverses the data every 4 bytes at offsets: $x, x+4, x+8, x+12$.

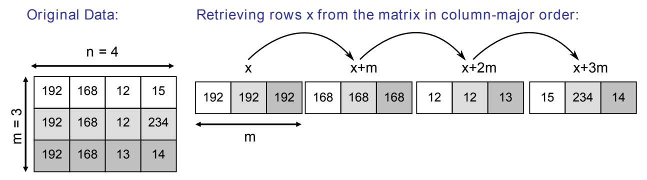


Figure 8: The row at position x from a 3×4 matrix stored in column-major order is retrieved by accessing the four bytes at the offsets $x, x+m, x+2m, x+3m$.

4.2 Partial-Decompression (PD)

The Full-Decompression decompresses all RLE-subblocks, even if they contain no results. Partial-Decompression decompresses only subblocks that contain query results. An illustration of this approach is given in Figure 9.

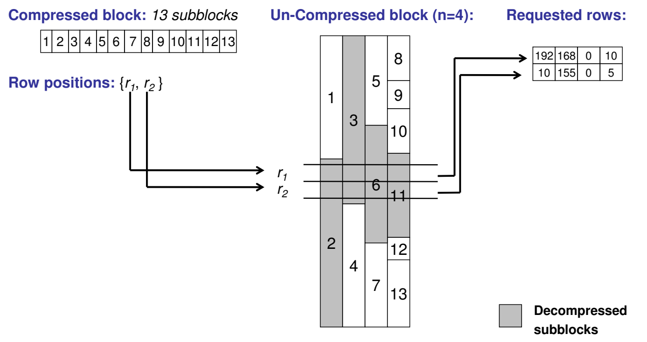


Figure 9: Partial Decompression (PD) decompress only RLE subblocks containing at least a byte that is required for retrieving the requested rows.

The partial decompression algorithm first calculates the set of offsets O in the matrix (stored in column-major order) that should be accessed to retrieve the requested rows. The offsets O are used to determine whether an RLE-subblock (`V-` or `B-block`) must be decompressed: a subblock b that compresses the matrix byte offset i to j needs to be decompressed if there is an offset $k \in O$, where $i \leq k \leq j$. The decompressor can efficiently find which subblocks need to be decompressed using the procedure we outline below. After decompressing the required subblocks, the requested rows are extracted using the same byte-traversal technique described for the Full-Decompression. The process described above is summarized in Algorithm 2.

Implementing Partial-Decompression: A naive approach to determine which RLE-subblock needs to be decompressed is to test, for each subblock, whether it contains at least one byte of the rows in the query (line 6 of Algorithm 2). For this purpose, we need to compute the *subblock range*, which is the range of byte offsets stored in an RLE subblock. If the span of a subblock is σ and the sum of the

Algorithm 2 PartialDecompress($Rows, B, n$)

```
Input:  A list of row positions $Rows$
        A compressed block $B$
        The size $n$ of each attribute value
Output: The attribute values stored in the requested row positions

1: $O := \emptyset$        ◁ start with an empty set of offsets

2: foreach $r \in Rows$        ◁ build the set of offsets
3:   $O := O \cup$ getOffsets($r$, $n$)

4: foreach $b \in SubBlocks(B)$        ◁ linear subblock scan
5:   $(i, j) :=$ subBlockRange($b$)
6:   if $\exists\ k \in O : k \leq j \wedge k \geq i$ then
7:       decompressSubBlock($b$, $i$, $output$)
8:   end if
9:       {Byte-traversal technique as in Figure 8}
10: return retrieveRowsFromColunMajor($Rows$,$output$,$n$)
```

spans of the preceding subblocks is S, then the range of a subblock is $[S+1, S+\sigma]$. Recall that the RasterZip encoding allows to efficiently compute the span of a subblock. In order to test whether a sublock range contains a byte of a queried row, the naive approach would require to map the offsets of the queried row bytes into the transposed matrix and then to test each subblock. If at least one of the new offsets lies in the range between $S+1$ and $S+\sigma$, then the corresponding subblock should be decompressed. This naive strategy is very simple, but requires many comparisons. In practice, it leads to very slow performance. Even if only a small fraction of rows have to be extracted, the decompression speed is considerably worse than performing a full-decompression.

We next describe a practical and efficient implementation of Algorithm 2 that uses a fixed amount of memory to store the set of offsets and substantially reduces the number of needed comparisons. The main idea of the implementation is to represent the set of offsets as a bitmap instead of a list of integers. Each bitmap entry corresponds to a certain range of bytes. A bitmap entry is set to 1 if and only if at least one byte in the corresponding range belongs to one of the queried rows. In the next paragraph, we describe the technique in more detail.

The transposed $m \times n$ matrix (in column-major order) is partitioned into s fixed size cells, where s is a small constant. Intuitively, a grid of s slots is logically superimposed on the matrix. A bitmap of length s, called *grid bitmap*, is used to mark each cell that contains at least one byte that is part of the query result (see Figure 10 where $s = 16$). Given the set of rows to be retrieved, the bits that need to be set to 1 can be determined very efficiently. First, the offsets of the desired row bytes are mapped into the transposed matrix. The mapped offsets are (still) in the range $[1, m \cdot n]$. Then, by linearly rescaling the offsets in the range $[1, s]$ we find the positions of the grid bitmap that need to be set.

Given the grid bitmap, it can be determined efficiently whether a subblock needs to be decompressed: its subblock range simply needs to be linearly rescaled as well by a factor of $s/(mr)$. If any bit is set in the grid bitmap in this rescaled range, the block needs to be decompressed. This process can exploit the bit counting assembly instruction and does not need any comparisons. The number s of cells is a tunable parameter that allows to trade off precision for performance. More detailed grids (i.e. with larger values of s) are more

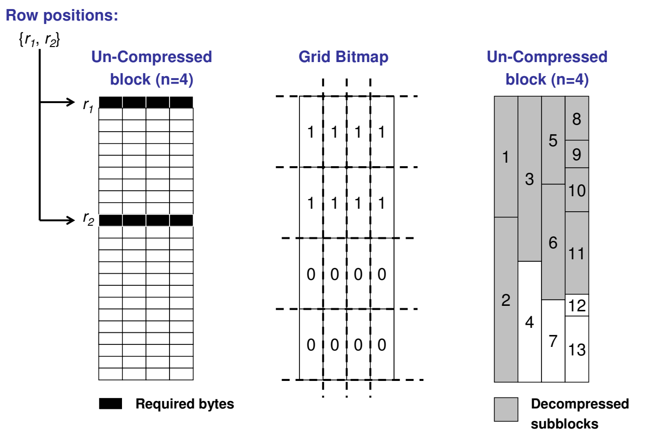


Figure 10: A Grid Bitmap is used to partition the uncompressed $m \times n$ matrix into 16 distinct cells ($s = 4 \times 4$). Given a query, the bit corresponding to each cell is set to 1 if there is at least a byte in the cell that has to be retrieved from the compressed data. A subblock has to be decompressed if and only if it overlaps with a cell that has been marked with a 1 in the corresponding Grid Bitmap.

precise in estimating the subblocks to be decompressed, but require more space to be stored, and consequently, are slower to query in average due to cache misses.

4.3 Adaptive Block Decompression

We have so far described two algorithms for block decompression. In this section we examine how to *adaptively* choose between the two algorithms at runtime. Partial decompression is preferred when a small fraction of RLE-subblocks need to be decompressed. Otherwise, the cost for computing the span of multiple RLE-subblocks makes full decompression a better strategy.

We introduce an adaptive decompression component that uses two runtime parameters, namely the *selectivity* of a query (within a block) and the *compression ratio* of a block, to predict which decompression strategy will result in faster decompression. Below, we describe how the two parameters relate to the decompression strategy:

- **Selectivity**: If the selectivity within a block is low (i.e., a large fraction of rows have to be retrieved), then many RLE subblocks need to be decompressed. In this case, full-decompression is preferable. In contrast, for high selectivity queries, a small fraction of subblocks is decompressed. In this case partial-decompression is faster.
- **Compression ratio**: The compression ratio is defined as the ratio between the size of the compressed data and the size of data before compression [23]. When the compression is better (lower compression ratio), then a block is compressed into fewer RLE-subblocks, and a larger fraction of the subblocks needs to be decompressed. Therefore, better compressed blocks benefit more from full decompression. Conversely, when RasterZip achieves worse compression ratios, a small fraction of RLE-subblocks have to be decompressed.

The selectivity of a query and the compression ratio of a block largely determine the fraction of RLE-subblocks that need to be decompressed and therefore can be used to predict which of the two strategies will perform better. In addition, the selectivity (s) and compression ratio (c) can be *efficiently computed at runtime before accessing the compressed subblocks*. We exploit these two parameters to build an *adaptive decompression component* shown in Figure 11. The adaptive decompression component accepts as input a compressed block b and a set R of row positions and, then, it uses a model to decide if partial or full decompression will provide a lower response time.

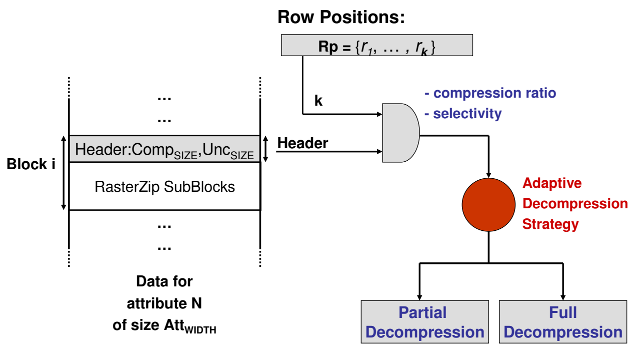


Figure 11: The adaptive decompression component decides at runtime the more efficient decompression strategy (partial or full) to be used for answering a query where k rows of the data block have to be accessed. It accepts as input the row selectivity s and the block compression ratio c.

Given a block b, the compression ratio can be efficiently computed from the block header, which stores the size of the block before and after compression (Unc_{SIZE} and $Comp_{SIZE}$ in Figure 11). The selectivity is computed by dividing the number of rows to be decompressed by the total number of attribute values packed in b. The adaptive decompression component, chooses between the two strategies using a simple and yet effective model built from profiling RasterZip. The model is a $l \times l$ matrix, where the two dimensions are selectivity and compression ratio. The value in a cell indicates the decompression strategy that is expected to provide faster decompression for the corresponding range of selectivity and compression ratio values.

A profiled model is data- and attribute-agnostic: once built using real or synthetic data, it can be applied for different datasets and queries. To profile RasterZip we executed queries of different selectivity on compressed traffic flow records and measured the time required to decompress the queried rows using both the partial and full decompression strategies. For each cell of the model matrix, we computed the average response time for the two decompression strategies. We finally selected the decompression strategy with the lowest average decompression time.

An actual example of the model is shown in Figure 12. Note that partial-decompression is preferable for blocks that exhibit poor compression and/or high selectivity. For the remaining cases, full-decompression should be chosen. In summary, the adaptive decompression component allows our methodology to combine the power of both full- and partial-decompression strategies.

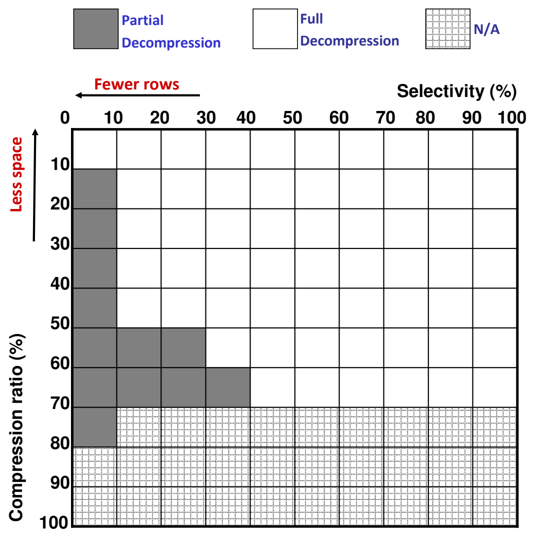


Figure 12: A 10×10 grid model built using a profiling dataset for deciding when to decompress using Partial-Decompression (PD) or Full-Decompression (FD). We indicate with N/A, cells that were not filled using the profiling dataset. Partial-Decompression is preferable for blocks that exhibit low compression ratios and/or high block selectivities.

4.4 Error detection

Contrary to block-based compression utilities, such as *bzip2*, RasterZip does not define a framing format and does not come with built-in support for error detection. In fact, RasterZip delegates this task to the upper layers where error detection mechanisms can be implemented using well-known approaches.

For example, RasterZip-encoded data blocks can be interleaved with a fixed size bit pattern, the block delimiter, that allows data block boundaries to be recognized and a checksum can be prepended to each RasterZip-encoded block. Data corruption can be detected by comparing the stored checksum with the checksum computed over the data decompressed using the full decompression algorithm. This approach is not applicable when the partial decompression strategy is used. However RasterZip could be modified to include an additional checksum at the end of `V-` and `B-blocks`. We leave this extension to future work.

5. EVALUATION

To evaluate the performance of RasterZip we built the archival solution shown in Figure 13 and use it to archive and query traffic flow records collected from real networks. The streaming flow records (13.A) are hashed into hash-buckets containing chains of 'similar' records (13.B). Data blocks are created from each hash chain and then indexed (13.C) and archived (13.D) in compressed data blocks using a columnar-approach. At query time, the index returns the row positions of interest in the archive. This information is used by the query processor to identify for each requested attribute the compressed data blocks within the corresponding data column that contain the result set.

For our comparison experiments, we compress attribute blocks in the component D of Figure 13 both with LZO and RasterZip. In this manner, both LZO and RasterZip benefit from the online record rearrangements provided by the component 13.B. The reordering is realized using oLSH, a flow record sorting methodology that we introduced in our previous work [14].

We choose a block size of 4000 records, which we found empirically to provide the best trade-off between compression ratio and response time. The indexing (13.C) is realized using the COMPAX compressed bitmap indexing encoding introduced in our previous work [14]. The only evaluation metric that depends on the selected index is the query response time. As shown in our previous work, the index lookup makes a negligible fraction of the query response time, which is dominated by the time for decompressing blocks even when a high-speed compressor such as LZO is used. In addition, the evaluated compressors are used with the same index to yield a fare comparison.

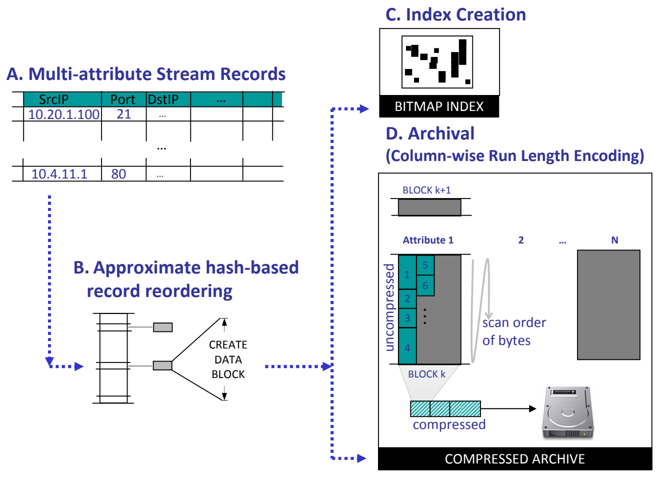


Figure 13: An overview of the compression methodology for streaming records.

Our experiments show that the column-major RLE-based compression significantly improves the compression ratio achieved with LZO. Additionally, because RasterZip provides *subblock decompression* it has better response times for high selectivity queries.

All experiments have been conducted on a commodity desktop machine equipped with 2 GB of DDR3 memory and an Intel Core 2 Quad processor (Q9400) running GNU/Linux (2.6.28 kernel) in 32-bit mode. The processor has four cores running at 2.66 GHz and 6 MB of L2 cache. We store the compressed archives on a 320 GB desktop hard drive[4].

As input to the system we provided uncompressed flow traces stored on a commodity solid state drive[5]. The drive provides a sustained reading speed of 170 MB/s, which corresponds to more than 5 Million flows/second (f/s). The system has been configured to fetch flows sequentially from the solid state drive and to store both indexes and compressed columns to the mechanical desktop hard drive. This simple setup allows us to reproduce flow rates that can only be observed in large ISP networks. We use two data sets in the evaluation:

- Six days of NetFlow traces of access traffic from a *large hosting environment* (HE).
- A two month NetFlow trace of internal and external traffic in an medium-sized enterprise *production network* (PN).

Each flow record includes the following attributes: source and destination IP addresses and L3 ports, the L3 protocol, TCP flags, source and destination Autonomous System (AS) number, number of packets, number of bytes, the time of the first packet and the duration of the flow. Information about the datasets are summarized in Table 1. The *gzip* and *bzip2* columns report the storage footprint when compressing the *raw* data with the *gzip* and *bzip2*, respectively.

Table 1: Utilized Datasets

Data	# flows	Length	Raw	*gzip*	*bzip2*
HE	231.9 M	6 days	6.9 GB	2.5 GB	2.2 GB
PN	1.2 B	62 days	37 GB	8.1 GB	6.9 GB

5.1 Compression Ratio

In Table 2, we report the size of the archives compressed with RasterZip and LZO. First, we note that the approximate flow reordering scheme allows LZO, which is optimized for compression speed rather than for achieving high compression ratios, to almost match the compression ratio of *gzip* (shown in Table 1). By properly exploiting the prefix structure of partially reordered traffic records, RasterZip can further reduce the disk consumption for both datasets. In fact, we observe that RasterZip uses 22% and 24% less space than LZO for the HE and PN datasets, respectively. It also uses less space than the simple approach of compressing raw data with *gzip* or *bzip2*. Compared to the values in Table 1, RasterZip offers 20% compression than *gzip* and even 9-10% better compression than the slow, but heavily space-optimized *bzip2*.

Table 2: Disk space requirements for different compression algorithms.

Dataset	LZO	RasterZip
HE	2.6 GB	2.0 GB
PN	8.2 GB	6.2 GB

5.2 Insertion Rate

Our system has been designed for compressing and indexing high-speed streams of flow records in real-time. RasterZip aims at reducing the disk usage of compressed archives and at supporting very high insertion rates. Having showed that RasterZip reduces the disk usage of two reference compressors, we evaluate the insertion rate it can support. For this purpose, we use the two datasets and compare the insertion rate our archive achieves when data columns are compressed on-the-fly using LZO and RasterZip.

[4]The hard drive is a 7200 rpm Hitachi HDP725032GLA380 equipped with 8 MB of cache. The system is capable of performing cached reading at 2400 MB/s and unbuffered disk reads at 80 MB/s (measured with `hdparm`).
[5]Intel X-25M G1, 80 GB model

Table 3: Record processing rates when using RasterZip or LZO for compressing data columns.

Dataset	LZO	RasterZip
HE	474K f/s	471K f/s
PN	513K f/s	512K f/s

As shown in Table 3, RasterZip realizes almost identical insertion rates with LZO, which is the fastest known compressor of the reference Lempel-Ziv family. The average insertion rate is close to 500 thousand flows/second (f/s) for both datasets regardless of the compression algorithm used. To put this number into perspective, medium-large networks exhibit peak rates of 50 thousand f/s. Therefore, we learn that RasterZip substantially reduces the disk consumption of LZO (and *gzip*) and supports very high insertion rates.

5.3 Adaptive Decompression Selection

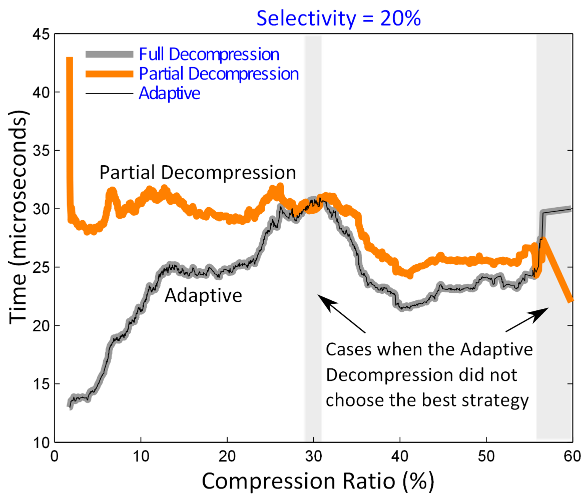
Next, we evaluate the adaptive decompression component to see if it can properly select between the full and partial decompression strategies at run-time.

Profiling stage: We constructed the model for the adaptive decompression component by profiling the decompression speed of the two decompression strategies using real network data and queries of different selectivity. Specifically, we created a set P containing the distinct IP addresses present in one hour of traffic from the production network dataset (PN). Then, we performed a query for every element in P and for every data block that was decompressed to answer the query we computed the following four metrics: the compression ratio of the block (percentage), the selectivity of the query (percentage), and the time required to decompress the block using full and partial decompression. We used these metrics to profile a very compact grid model of size 8×8 bits as described in Section 4.3.

Evaluation stage: We evaluate the adaptive decompression component by comparing the block decompression time it provides with the decompression time offered by the full and partial decompression strategies. For this purpose, we issue the IP queries described above over a different time frame. In Figure 14 we illustrate the query response time using full, partial, and adaptive decompression. The figure on the bottom shows how the response time varies with selectivity for a fixed compression ratio (20%), while the one on the top shows how the response time varies with the compression ratio when the selectivity is fixed to 20%. The line corresponding to adaptive decompression is at the bottom for the vast majority of the queries. This illustrates that the adaptive decompression component effectively selects the decompression strategy that is faster. This is feasible simply by relying on a very compact model stored in a $8{\times}8$ bitmap. We use this model for the rest of our evaluation experiments.

5.4 Overall Performance

Finally, we measure the overall query response time of the archives compressed using RasterZip with the adaptive decompression component enabled. We compare the response time of this setup with the response times that the system offers when LZO is used for data compression. Note that the response time for answering queries, as we have shown in our previous work [14], is dominated by the time spend in decompressing data blocks rather than by the time spend in looking up the compressed bitmap indexes to find blocks that need to be decompressed.



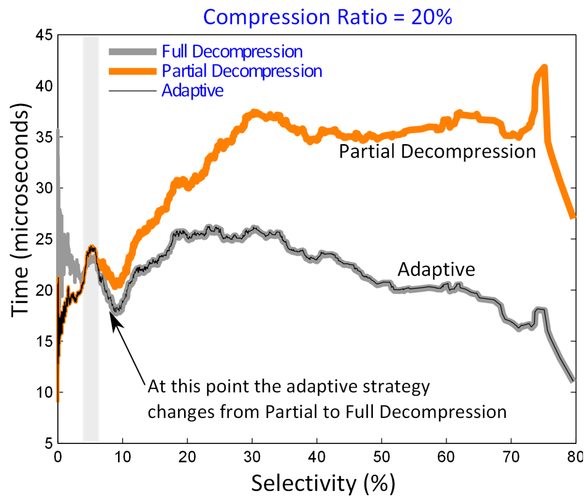


Figure 14: The Adaptive strategy chosen on-the-fly intelligently shifts between Full-Decompression and Partial-Decompression. Shown for a fixed compression ratio (bottom) and selectivity (top).

To compare the two compressors, we query for flow records using ports corresponding to well-known applications.[6] For this purpose, we extracted the 311 ports listed in the file */etc/services* of the test machine and executed destination port queries over 7 days of each dataset (PN and HE). This corresponds to 2,177 different queries, each selecting all the existing flow records attributes (i.e.,`"SELECT * FROM flows WHERE dstPort=X"` in SQL parlance). Note that using single- or multi-attribute queries, e.g., `dstPort=`X `AND dstIP=`Y, does not play a role in the comparison between LZO and RasterZip because it would only affect the index lookup time of a query equally for both compressors. What is important is to compare the two compressors over a range of query selectivity values. Varying the selectivity can be done in multiple ways. In our experiments, we selected to do this by issuing single-attribute queries as this is a simple

[6]The Internet Assigned Numbers Authority (IANA) assigns port numbers to applications.

way to do it. We ordered queries by the number of rows retrieved (the selectivity) and, for each query, we plot in Figures 15 and 16 the ratio between the response time when the query is executed over a RasterZip compressed archive and the response time for the same query executed over an LZO compressed archive. Values lower than one correspond to queries where RasterZip offers better response times than LZO.

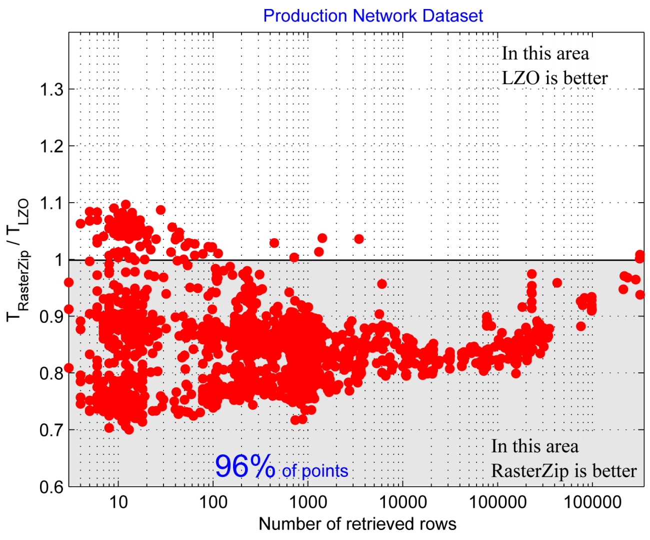


Figure 15: RasterZip over LZO query response time for the Production Network dataset (PN).

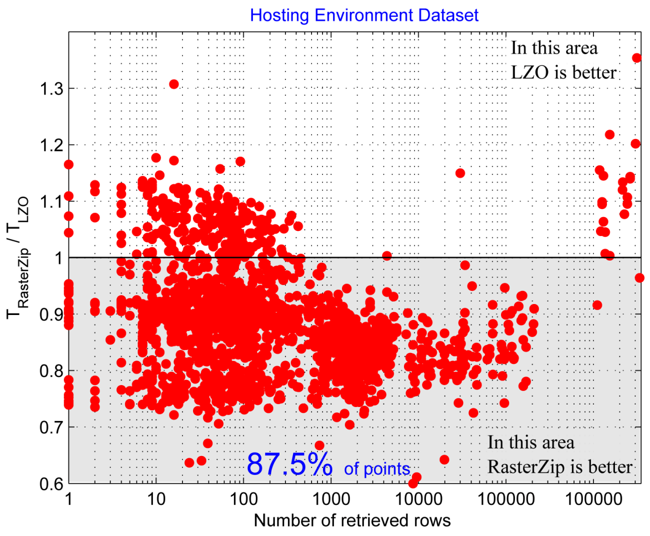


Figure 16: RasterZip over LZO query response times for the Hosting Environment dataset (HE).

RasterZip provides better query response time than LZO for more than 87% − 96% of the queries depending on the dataset (see Figure 15 and 16 for the PN and HE dataset, respectively). For high selectivity queries, the query response time is in the order of milliseconds. The few cases where RasterZip offers worse response time than LZO correspond to: a) a small fraction of the high selectivity queries (left top side of the two figures) and b) to extremely low selectivity queries. Examples of the second type of queries can be observed on the extreme right side of Figure 16, which is based on the data set from the hosting environment. These points correspond to queries on port 80 (HTTP protocol), which, in the hosting environment, match an extremely large fraction of the flows. Such queries of extremely low selectivity are not typically useful for network administrators, since they match too many flows. Our system is optimized for high selectivity queries, i.e., searching for "needles in a haystack", which are shown in the middle and left side of Figures 15 and 16.

6. RELATED WORK

General-purpose compression has been extensively studied in the previous decades, which led to a set of state-of-the-art algorithms and associated utilities that are presently extensively-used. The well-known Lempel-Ziv family of algorithms [28] includes *gzip* and LZO, which is the fastest member of the family [27]. Due to the high compression and decompression speed, LZO is a compressor widely used to accelerate I/O bound data-intensive workloads (e.g., in Hadoop clusters [19]).

Although the available compression gain margins have become extremely tight, in this work we show that by exploiting the patterns of network traffic data we can improve the compression ratio of LZO (and also of *gzip* and *bzip2*) by a significant fraction of 22-24%, while reducing query response times and without incuring any penalty. The RadixZip Transform [24] is similar to our approach in that it transposes bytes by the sort order of their prefixes to boost the compression of token based streams, such as urls and logs. RadixZIP offers up to 20% better compression ratios and up to 10% higher decompression speed than *bzip2*, the widely used block based compressor based on the Burrows-Wheeler Transform (BWT). In contrast with RadixZip and *bzip2*, which are optimized for achieving high compression ratios, RasterZip is optimized for speed and achieves much faster (de)compression speeds. Additionally, it has been designed to support fine-grained partial decompression operations, which are hard to accommodate in *bzip2* even with the help of an index. In fact, *bzip2*-compressed data is made of data blocks that can be independently decompressed. However, the block size is large (between 100Kb and 900Kb), and as a result, even highly selective queries can require the decompression of the entire archive.

Silk [15], nfdump [17], and flowtools [22] are commonly-used tools for storing streams of flows. They all store flow records as flat files, which are eventually compressed using general purpose compressors and compression algorithms, such as *gzip*, *bzip2*, and LZO.

Recently, compression algorithms have been used to boost the performance of columnar databases. A comparison of the performance of different general-puropose compression algorithms for columnar databases can be found in [2]. In our previous work [14], we built a columnar database that uses a novel indexing scheme optimized for network flow records and the LZO general-puropose compressor. In this, work we visit the subject of compression and introduce a domain-optimized compression scheme that provides fine-grained decompression granularity and high compression efficiency. Recently, Giura *et al* [16] optimized a columnar database called NetStore for storing flows. NetStore provides the capability to select, at runtime, among different standard compression tools. However, when deployed on a significantly less powerful machine (4 cores instead of 8, 2 Gb of RAM instead of 6 Gb), our solution provides insertion rates that are two orders of magnitude higher.

7. WHAT IS THE COST OF STORAGE?

The costs for storing and maintaining large repositories is commonly underestimated, mostly because the price of desktop hard drives dropped significantly in the last few years. Unfortunately, the *disk space is cheap* motto only holds at small scales. Historical network traffic repositories can be massive. For example, large traffic archives maintained by academic and research institutions, like the flow record repository of the Communication Systems Group at ETH and the UCSD Network Telescope data of CAIDA, are both close to 100 Terabytes [4]. Organizations, like medium and large ISPs, cloud providers, and law enforcement agencies, monitor much more traffic and need to handle even bigger traffic data. Large-scale repositories must be highly available and fault-tolerant to prevent data loss. Fault-tolerance can be achieved by replicating data in a distributed setting (e.g., the Google approach) or by using enterprise class storage products (e.g, disk arrays with RAID support). In both cases, the operational costs include additional expenses for energy consumption and for performing backups over tape drives. Based on current storage prices [5], the cost for storing 100 Terabytes is, for example, between $40,000 and $72,000 per year for different types of service. High storage costs prevent the wider availability of rich Internet measurement data, and are, in fact, the sole reason why even existing traffic repositories that have been very influencial in studying Internet threats are threatened with deletion [4]. A compression technology that can substantially reduce data volume has a direct and very tangible economical impact, which can be measured in several thousands of dollars per year for few dozens of Terabytes. RasterZip, by compressing more than 20% better than *gzip*, reduces costs accordingly.

8. CONCLUSION

In this work we introduce RasterZip, the first high-performance compressor specifically designed for network traffic archives that exploits the prefix structure of partially reordered data to provide fine-grained decompression granularity. RasterZip supports online high-speed stream compression, enables interactive queries over archived data, leverages network data patterns, and exploits features of modern hardware. In addition, we introduce a novel decompression scheme that provides fine-grained decompression granularity at the attribute level in order to accelerate queries for "needles in a haystack". Our evaluation shows that RasterZip outperforms general-purpose compressors on traffic flow data from two different networks. Compared to the widely-used state-of-the-art *lzop* compressor, RasterZip accelerates queries, reduces expenses for storage by 22-24% due to more compact compression, and does not incur penalty in any other performance metric.

9. REFERENCES

[1] StealthWatch FlowCollector. http://www.lancope.com/products/stealthwatch-flowcollector/.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proc. 32nd ACM SIGMOD Int. Conf. on Management of Data*, pages 671–682, 2006.

[3] E. W. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu. Accelerating Network Traffic Analysis Using Query-Driven Visualization. In *Proc. of IEEE Symposium on Visual Analytics Science and Technology*, pages 115–122, 2006.

[4] CAIDA. Targeted serendipity: the search for storage. `http://blog.caida.org/best_available_data/2012/04/04/`, 2012.

[5] S. D. S. Center. Sdsc project storage pricing options. `http://project.sdsc.edu/pricing.php`.

[6] X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

[7] B. Claise. RFC 3954: Cisco Systems NetFlow Services Export Version 9, 2004.

[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proc. of the 20th Annual Symposium on Computational Geometry*, pages 253–262, 2004.

[9] L. Deri, V. Lorenzetti, and S. Mortimer. Collection and Exploration of Large Data Monitoring Sets Using Bitmap Databases. In *Proc. of the 2nd Int. Workshop on Traffic Monitoring and Analysis*, TMA'10, pages 73–86, 2010.

[10] N. Duffield, C. Lund, and M. Thorup. Charging from Sampled Network Usage. In *Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW)*, pages 245–256, 2001.

[11] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. *IEEE/ACM Transactions on Networking (ToN)*, 9:265–280, 2001.

[12] A. Friedl and S. Ubik. Perfmon and Servmon: Monitoring Operational Status and Resources of Distributed Computing Systems. Technical Report 10, CESNET, Prague, Czech Republic, 2008.

[13] F. Fusco, X. Dimitropoulos, M. Vlachos, and L. Deri. pcapindex: an index for network packet traces with legacy compatibility. *SIGCOMM Comput. Commun. Rev.*, 42(1):47–53.

[14] F. Fusco, M. Vlachos, and M. P. Stoecklin. Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*, 21(3):287–307, June 2012.

[15] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas. More Netflow Tools for Performance and Security. In *Proc. of the Conf. on Large Installation Systems Administration*, pages 121–132, 2004.

[16] P. Giura and N. Memon. Netstore: an efficient storage infrastructure for network forensics and monitoring. In *Proc. of the 13th Int. Conf. on Recent advances in intrusion detection*, RAID'10, pages 277–296, 2010.

[17] P. Haag. NFDump. http://nfdump.sourceforge.net/.

[18] R. Hofstede, A. Sperotto, T. Fioreze, and A. Pras. The network data handling war: MySQL vs. NfDump. In *Proc. of the 16th EUNICE/IFIP Conf. on Networked services and applications: engineering, control and management*, EUNICE'10, pages 167–176, 2010.

[19] S. B. Joshi. Apache hadoop performance-tuning methodologies and best practices. In *Proc. of the 3rd joint WOSP/SIPEW Int. Conf. on Performance Engineering*, ICPE '12, pages 241–242, 2012.

[20] A. Lakhina, M. Crovella, and C. Diot. Mining Anomalies Using Traffic Feature Distributions. In *Proc. ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 217–228, 2005.

[21] T. W. Lam, W.-K. Sung, S.-L. Tam, C.-K. Wong, and S.-M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

[22] S. Romig, M. Fullmer, and R. Luman. The OSU Flow-tools Package and Cisco NetFlow Logs. In *Proc. Conference on Large Installation Systems Administration (LISA)*, pages 291–303, 2000.

[23] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 2nd edition, 2000.

[24] B. D. Vo and G. S. Manku. RadixZip: Linear Time Compression of Token Streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 1162–1172, 2007.

[25] A. Wagner. *Entropy-Based Worm Detection for Fast IP Networks*. PhD thesis, ETH Zurich, 2008.

[26] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proc. of the 32nd ACM SIGIR Conf. on Research and development in information retrieval*, SIGIR '09, pages 147–154, 2009.

[27] L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar. Online memory compression for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9:27:1–27:30, 2010.

[28] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.