TerseCades: Efficient Data Compression in Stream Processing

Gennady Pekhimenko
Chuanxiong Guo, Myeongjae Jeon, Peng Huang, Lidong Zhou
Huge volumes of streaming data with real-time processing requirements
Enormous pressure on the capacity and bandwidth of servers’ main memory
Is Data Compression Useful for Streaming?

• Intuitively, streaming with simple operators should be *bandwidth-bottlenecked*: either network or memory bandwidth

• Simple single node experiment with the state-of-the-art streaming engine, Trill, with the *Where* query over large one column 8-byte field: E.g., Where (e => e.errorCode != 0)

• Expectation: observe *memory bandwidth* as a major bottleneck
Compressibility $\not\implies$ Performance Gain

Ideal 8X Compression vs. No Compression on Where Query with Trill

![Graph showing throughput comparison between ideal 8X compression and no compression across different thread counts.]

- Only 10%-15% performance improvement with 8X compression
What Went Wrong?

❌ Memory allocation overhead:
  just-in-time copy of payloads to create a streamable event

❌ Memory copying and reallocation:
  enables flexible column-oriented data batches

❌ Inefficient bit-wise manipulation

❌ Hash tables manipulations
Compressibility => Performance Gain

8X Compression with Add benchmark from STREAM suite

Throughput (MElems/sec)

# Threads

Char CharCompr. Long

8X Compression with Add benchmark from STREAM suite

Up to 6.1X speedup with realistic compression algorithm: Base-Delta-Encoding

If no artificial bottlenecks: performance improvement is close to compression ratio (9.6X speedup with 8X compression)
Prerequisites for Efficient Data Streaming

✓ Fixed Memory Allocation

✓ Efficient HashMap Primitives

✓ Efficient Filtering Operations (bit-wise manipulations)
Key Observations

• Memory bandwidth becomes the major bottleneck if streaming is properly optimized

• Dominant part of the data is synthetic in nature and hence has a lot of redundancy
  – Can be exploited through efficient data compression
TerseCades: Baseline System Overview
Key Design Choices and Optimizations

✓ **Lossless Compression**
  ✓ Arithmetic vs. Dictionary-based Compression
  ✓ Decompression is on the critical path
✓ **Lossy Compression without Output Quality Loss**
  ✓ Integers and floating points
✓ **Reducing Compression/Decompression Cost**
  ✓ Hardware-based acceleration: vectorization, GPU, FPGA
✓ **Direct Execution on Compressed Data**
Lossless Compression: Base-Delta Encoding

- Fast Decompression: vector addition
- Simple SW HW Implementations: arithmetic and comparison
- Effective: good compression ratio
Lossy Compression Without Output Quality Loss

• Base-Delta Encoding modification
  – Truncate deltas when full precision not required

• ZFP floating point compression engine
  – Equivalent of BD in floating point domain with controlled precision
Reducing Compression Overhead

SIMD/Vectorization
- Intel Xeon with 256-bit SIMD

GPU
- NVIDIA 1080Ti

FPGA
- Altera Stratix V
Execution on Compressed Data

- **Compress**: Performs compression to reduce data size.
- **Decompress**: Performs decompression to restore data to its original form.

- **Memory** stores compressed data.
- **Processor** interacts with memory and performs compression and decompression tasks.

- **Issues**:
  - Incurs decompression and compression latency.
  - High energy overhead.

**Question**:
Can we leverage data being in a condensed form?
Execution on Compressed Data

Key 1 | Value 1
---|---
Key 2 | Value 2
Key 3 | Value 3
Key N | Value N

Memory

Value 1 | Value 2 | Value 3 | Value N
---|---|---|---
8B | 8B | 8B | 8B
Execution on Compressed Data

- **Value 1**
- **Value 2**
- **Value 3**
- **Value N**

- **8B**

- **Value**

- **Meta data**
- **Value 1**
- **Value 2**
- **Value 3**
- **Value N**

- **1B**

**N 8-byte Comparisons**

**1 or N/8 Comparisons**

- ✓ **Low Latency**
- ✓ **Single Comparison**
- ✓ **Narrower Operations**

16
Evaluation: Methodology

• CPU: 24-core system based on Intel Xeon CPU E5-2673, 2.40GHz with SMT-enabled, and 128GB of memory

• GPU: NVIDIA GeForce GTX 1080 Ti with 11GB of GDDR5X memory

• FPGA: Altera Stratix V FPGA, 200MHz
STREAM Benchmark Results

Add benchmark from STREAM suite

Vectorization further reduces compression/decompression overhead, especially for smaller number of threads
When direct execution is applicable, it can significantly improve performance as it reduces the total computation.
Monitoring and Troubleshooting: PingMesh

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

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())

<table>
<thead>
<tr>
<th>TimeStamp (8, BD)</th>
<th>ErrorCode (4, EN+BD)</th>
<th>SrcCluster (4, HS+BD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DstCluster (4, HS+BD)</td>
<td>RoundTripTime (4, BD)</td>
<td></td>
</tr>
</tbody>
</table>

BD – Base+Delta encoding
HS – String hashing
EN – Enumeration

Number in parenthesis – number of bytes before compression
PingMesh C2cProbeCount Results

Total of more that 15X improvement in throughput due to data compression with efficient optimizations
The highest performance benefits are for operators where direct execution is applicable (e.g., Where)
### IaaS VM Performance Counters

<table>
<thead>
<tr>
<th>TimeStamp (8, BD)</th>
<th>Cluster (11, HS)</th>
<th>VmID (36, HS)</th>
<th>SampleCount (4, BD)</th>
<th>MinValue (8, ZFP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxValue (8, ZFP)</td>
<td>CounterName (15, EN)</td>
<td>NodeId (10, HS)</td>
<td>Datacenter (3, HS)</td>
<td>AverageValue (8, ZFP)</td>
</tr>
</tbody>
</table>

BD – Base+Delta encoding; HS – String hashing; EN – Enumeration; **ZFP** – efficient floating point compression (lossy with controlled accuracy)

Number in parenthesis – number of bytes before compression

**Upto 6X compression with ZFP lossy compression algorithm**
IoT Datasets

• **Geolocation data** (GPS coordinates from GeoLife project):
  – 4.5X average compression ratio
  – Less than $10^{-6}$ loss in accuracy

<table>
<thead>
<tr>
<th>TimeStamp (8, BD)</th>
<th>Latitude (8, ZFP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longtitude (8, ZFP)</td>
<td>Altitude (4, BD)</td>
</tr>
</tbody>
</table>

• **Weather data** (Hurricane Katrina in 2005)
  – 3X-4X compression ratios for 18 metrics used in the data set
Comparison to Prior Work

• Compression in databases
  – Succinct, NSDI’15: execution on compressed textual data, complete redesign of data storage in memory
  – Abadi, SIGMOD’06: compression in column-oriented data stores; uses conventional compression algorithms not applicable to streaming

• Generic memory compression
  – Execution on compressed data is not supported
  – Lower compression ratios due to generality of algorithms chosen
Summary

• Q: Can **data compression** be effective in stream processing?
• A: **Yes**, our TerseCades design is the proof-of-concept
  – Properly optimize the baseline system
  – Use light-weight data compression algorithms + HW acceleration
  – **Directly execute** on compressed data
• Results on troubleshooting workload used in production allowed to replace 16 servers with just one!
TerseCades: Efficient Data Compression in Stream Processing

Gennady Pekhimenko
Chuanxiong Guo, Myeongjae Jeon, Peng Huang, Lidong Zhou