The Free Lunch Has a Queue: Characterizing On-chip Compression Accelerators for Analytics

Yi Jiang
EPFL
Lausanne, Switzerland
yi.jiang@epfl.ch

Antonio Boffa
EPFL
Lausanne, Switzerland
antonio.boffa@epfl.ch

Hamish Nicholson
EPFL
Lausanne, Switzerland
hamish.nicholson@epfl.ch

Anastasia Ailamaki
EPFL
Lausanne, Switzerland
anastasia.ailamaki@epfl.ch

ABSTRACT

Compression is a central mechanism for reducing data movement in analytical systems, but it also creates a resource-management problem: software compression consumes CPU cores, cache capacity, and memory bandwidth that the query engine needs for useful work. Modern server processors now provide on-chip compression accelerators such as Intel IAA and Intel QAT, yet their advertised throughput does not answer the DBMS-level question of when offload is actually the right execution path.

This paper characterizes CPU software, IAA, and QAT across database-relevant block sizes, codecs, and submission concurrency. We find that hardware offload is not a universal replacement for CPU compression. The best execution path depends on codec support, operation direction, block size, and accelerator queue state. IAA is a strong path for Deflate decompression, including small blocks where fixed overheads dominate other paths. QAT provides broader codec coverage, including LZ4 and Zstd compression, but is more sensitive to block size and batching. A full CPU socket remains competitive for some large-block compression cases, while also representing valuable query-processing capacity. The main implication is that compression should be scheduled dynamically: route blocks by codec, operation, size, and accelerator load; cap per-device submission concurrency; and fall back to software when offload is unsupported or saturated. Because these signals are coarse-grained and already available, the routing policy can be implemented as a lightweight per-block threshold rule rather than an expensive runtime optimization problem.

VLDB Workshop Reference Format:
Yi Jiang, Antonio Boffa, Hamish Nicholson, and Anastasia Ailamaki. The Free Lunch Has a Queue: Characterizing On-chip Compression Accelerators for Analytics. VLDB 2026 Workshop: 17th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS26).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yijiang23333/adms26_onchip_accel.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

As dataset sizes continue to grow, modern analytical database systems increasingly rely on heavily compressed data formats [1, 16, 21, 36, 45, 46, 48] to constrain storage footprints and reduce network I/O, thereby minimizing the data movement overhead across DRAM, local and remote NVMe, and remote object storage (e.g., Amazon S3 [13]). Software-based compression and decompression sit on the critical path of query execution.

Executing compression algorithms like Deflate [23], Brotli [12], LZ4 [9], or Zstd [51] on general-purpose processing cores is highly compute-intensive. Compression and decompression have recently been characterized as 14-31% of the datacenter tax [25, 31] for big data processing on hyperscale data stores, taking up a significant portion of the CPU cycles that can otherwise be used for core application logic. In a database context, it manifests as resource contention. When CPU cores are performing bulk software decompression, they stall query pipelines, contend for memory bandwidth, and pollute the last-level cache (LLC). Consequently, fewer resources are available for actual analytical operators (e.g., hash joins and aggregations), degrading query performance.

To alleviate this resource bottleneck, researchers have explored various software and hardware solutions. There have been many efforts to improve software algorithms, but they cannot fully recover the CPU cycles used for compression and decompression [14, 38, 42]. Therefore, researchers turn to hardware accelerators to offload the entire compression and decompression. While traditional off-chip, PCIe-attached accelerators (e.g., GPU [18, 35, 41], PIM [30], SmartNIC [29]) speed up de-/compression, traversing the PCIe bus and navigating kernel-level context switches, staging copies, and mandatory pinning erode those gains. Off-chip accelerators are especially ill-suited for fine-grained, microsecond-latency operations required by database buffer pools and transactional workloads.

To resolve performance interference without incurring PCIe latency penalties or being constrained by limited accelerator device memory, the hardware trend is shifting rapidly toward System-on-Chip (SoC) integration. Modern architectures, such as Intel Xeon 6 (Granite Rapids) [40], IBM Power9 and z15 processors [11], embed

hardware accelerators directly onto the CPU mesh. By integrating engines like the In-Memory Analytics Accelerator (IAA) and QuickAssist Technology (QAT), these architectures enable direct, cache-coherent access to the CPU's native memory subsystem. Furthermore, the introduction of shared virtual memory, direct cache access, and user-space dispatch instructions (e.g., `ENQCMD`) allows database engines to submit de-/compression tasks to the accelerators with sub-microsecond dispatch overhead [47]. By offloading the compute-intensive de-/compression to those on-chip accelerators, the CPU is freed from this work and can devote more cycles to the database engine's core data processing.

Although on-chip accelerators are increasingly used in DBMS, it remains unclear how systems should choose among CPU software, IAA, and QAT for compression and decompression. Prior work has studied IAA for database decompression [33] and CPU-less processing by pushing filters and joins down to the accelerator [15]. We complement these studies by focusing on a lower-level but unavoidable systems question: how compression work should be routed when it arrives in database-relevant units such as pages, column chunks, spill blocks, and storage-tiering fragments. Advertised accelerator bandwidth alone is not a sufficient basis for this decision because the best path depends on codec support, operation direction, block size, submission cost, and finite accelerator queue capacity.

In this paper we characterize CPU software, IAA, and QAT across database-relevant block sizes and submission concurrency. The experimental results lead to the following contributions and routing implications for analytical systems:

- ***Block granularity determines when compression offload pays off.*** IAA and QAT expose very different break-even points: IAA is effective for decompression even on small blocks, whereas QAT needs larger blocks or batching to amortize its API-level costs. Our characterization on 4 KB-2 MB block sizes shows that peak accelerator bandwidth is a poor predictor of the best execution path.
- ***Accelerator throughput is bounded by saturation, not by the number of submitters.*** A single IAA or QAT endpoint reaches peak throughput at modest submission concurrency; additional submitters do not create additional device capacity and may degrade throughput. This turns accelerator queues into first-class scheduling resources.
- ***The right de-/compression path is a scheduling decision for the query engine.*** Codec support, operation direction, block size, accelerator load, and CPU opportunity cost jointly determine whether a block should use IAA, QAT, or software. The resulting policy is hybrid: use local accelerators up to saturation, route unsupported or inefficient cases to the CPU, and avoid treating hardware offload as a universal replacement for software compression.

Our insights show that the compression tax is mitigated by effectively scheduling finite queue resources and managing microsecond-latency trade-offs rather than by maximizing device-level throughput.

2 BACKGROUND AND MOTIVATION

This section provides a foundation necessary to understand on-chip acceleration in analytical engines. We explain the computational overhead of general-purpose software compression, a major component of the "datacenter tax". Then we outline the heterogeneous block-size granularities typical of multi-tiered database storage engines. Finally, we contrast traditional, high-latency PCIe-attached hardware offloading with the emerging trend of cache-coherent, System-on-Chip (SoC) integration in modern server processors.

2.1 Datacenter tax in compression

Compression is no longer an optional storage optimization in modern analytical systems. As datasets exceed the capacity of DRAM and single storage nodes, database engines rely on lossless compression to reduce storage footprint, network traffic, memory pressure, and the cost of moving intermediate results across the memory and storage hierarchy. The most widely used general-purpose compressors in data systems are based on the LZ77 family [49, 50]. They reduce data size by identifying repeated byte sequences in a sliding window of previously processed input and encoding future occurrences as references to prior output. These references are typically represented through offset-length pairs, optionally followed by literals for bytes that cannot be matched.

The general-purpose compressors we evaluate differ in how they combine LZ77 and entropy coding. LZ4 keeps the representation simple and does not apply a separate entropy-coding stage, prioritizing compression and decompression speed [9]. Deflate compresses the resulting symbols with Huffman coding [39]. Zstd uses finite-state entropy coding from the ANS family for its sequence representation [24]. Deflate's Huffman step admits two variants: dynamic Huffman optimizes a fresh code table per block (two passes, best ratio, highest compression cost), while static Huffman reuses a precomputed table across blocks (single pass, slightly lower ratio, lower compression cost) [23, 39].

This reduction in bytes, however, introduces an execution cost. When compression and decompression run on general-purpose CPU cores, they compete with query execution for cycles, memory bandwidth, and cache capacity. In analytical workloads, this cost appears directly on the critical path: decompression must often precede scans, joins, and aggregations, while compression is required when spilling intermediate state, flushing cold pages, or moving data to lower storage tiers. Thus, the system saves space and reduces data movement, but pays the price of contended compute resources. This is the compression component of the datacenter tax: moving and storing bytes becomes cheaper, but query execution loses CPU time, memory bandwidth, and LLC residency to general-purpose compression work.

2.2 Block size granularity in databases

Data systems with storage tiering involve heterogeneous block sizes, each presenting distinct de-/compression speed and ratio requirements:

- **4KB – 16KB (Buffer Pool):** Fine-grained operations such as B-Tree node fetches [4, 27]. These are latency-sensitive: even sub-microsecond delays in user-space dispatching can degrade transactional throughput [43].

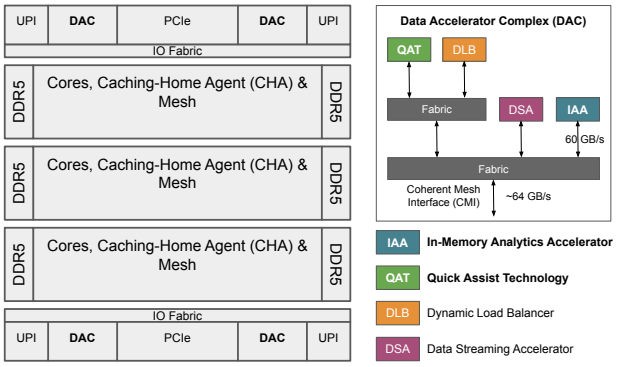

Figure 1: Intel *Granite Rapids* CPU Data Accelerator Complex (DAC) architecture overview.

- **64KB – 256KB (Warm Tier):** Columnar segments (e.g., Parquet row-groups) migrating between DRAM and local NVMe [32]. High throughput and moderate latency are required.
- **1MB+ (Cold Tier):** Deep archiving and S3 object flushes [3]. Compression ratio is paramount, and execution latency can be easily hidden via asynchronous background polling.

2.3 System-on-Chip acceleration

Unlike discrete PCIe cards, modern SoCs integrate endpoints directly into the coherent interconnect mesh. Intel embeds multiple accelerators into a built-in Data Accelerator Complex (DAC) on chip with their recent generations of Intel Xeon Scalable processors (*Sierra Forest, Granite Rapids*). A DAC (Figure 1) contains four different types of accelerators. Of these, the In-Memory Analytics Accelerator (IAA) and QuickAssist Technology (QAT) support de-/compression. Given the scope of this paper, here we focus on their de-/compression capabilities.

IAA is designed to accelerate big data workloads and in-memory analytics, supporting compression and decompression, analytics primitives, and data integrity functions. IAA's de-/compression is limited to Deflate.

QAT provides hardware acceleration for offloading security, authentication, and compression tasks, supporting data de-/compression and de-/encryption. It supports various compression algorithms, including Deflate, LZ4, and Zstd (compression only).

To integrate IAA and QAT into applications, Intel provides user-space libraries: QPL [5] for IAA, and QATlib [6] and QATzip [7] for QAT. These libraries convert application API calls into device descriptors and submit them from user space; the kernel driver only configures the device and grants access, while the device firmware parses each descriptor and drives the accelerator.

Both accelerators access application memory through shared virtual memory (SVM). Each issues memory requests that the IOMMU translates within the submitting process's virtual address space, so the device operates directly on the application's pageable memory and removes the staging copies and mandatory pinning that discrete accelerators require [47]. Intel Scalable I/O Virtualization (SIOV) builds on the same address-translation path to share a device across virtual machines, but our characterization targets bare-metal submission and does not exercise it.

The two accelerators differ in how software submits work. IAA reuses the Intel DSA work-submission interface [47]: software writes a 64-byte descriptor to a memory-mapped work-queue portal, using `ENQCMD` for a shared work queue or `MOVDIR64B` for a dedicated one. The QPL library we use exposes only the shared-queue path [5], where each `ENQCMD` returns an accept-or-retry status; multiple threads therefore submit to a single work queue without software locks and see back-pressure when it is full. QAT instead submits through descriptor rings in host memory [28]: the user-space library writes a request descriptor into a circular buffer and advances a tail pointer in a device register, after which the accelerator reads the descriptor over DMA and posts the result on a response ring. Both paths keep the kernel off the per-request data path. QAT incurs an MMIO tail update on publication, which it amortizes across batched requests, whereas IAA pays a fixed per-descriptor instruction cost; Section 3.2 measures how block size interacts with these submission costs.

3 CHARACTERIZATION

To understand how modern database systems should integrate System-on-Chip (SoC) accelerators, we first isolate their de-/compression performance. In this section, we present a characterization of the CPU, IAA, and QAT across multiple dimensions: storage block sizes, concurrency saturation limits, and algorithm-specific hardware constraints. This empirical foundation dictates how a query engine should offload de-/compression tasks.

3.1 Experimental methodology

System and Devices: Table 1 summarizes the specifications of our systems and the IAA and QAT accelerators. However, these maximum hardware data rates do not equate to actual end-to-end decompression throughput in query pipelines. We run all experiments on a single socket (96 physical cores) with NUMA-local memory. We disable hyper-threading and lock the CPU frequency for measurement stability. Unless otherwise specified, the performance of IAA and QAT is limited to a single device.

	Description
OS (kernel)	Ubuntu 24.04.4 (Linux kernel v6.17)
CPU	Intel(R) Xeon(R) 6952P CPU @ 2.1GHz
	2 sockets × 96 cores, hyper-threading disabled
Caches (total)	L1: 21 MiB, L2: 384 MiB, LLC: 960 MiB
Memory	Socket 0: 12 × DDR5-4800 channels
On-chip QAT	8 devices (4 per socket)
	160 Gbps (~20 GB/s) data bandwidth per QAT
On-chip IAA	8 devices (4 per socket)
	8 AEs and 16 WQs per IAA
	60 GB/s data bandwidth per IAA

Table 1: System configuration

CPU software: We use lzbench 2.2.1 [2] for the CPU software de-/compression baseline, compiled with GCC 13.3.0. We measure

de-/compression performance with 1 and 96 threads (physical cores) to test the base and peak performance of de-/compression within a single socket.

IAA: We use Intel Query Processing Library (QPL) [5] 1.9.0 to interact with IAA. It allows specifying the execution path as hardware (IAA), software (CPU host), or auto (automatic switching between IAA and CPU based on availability and load balancing). We measure de-/compression performance with 1 IAA device.

QAT: We use Intel QATzip [7] 1.3.2, and QAT-ZSTD-PLUGIN [8] 1.0.0 to offload de-/compression and measure the performance with 1 QAT device. We configure QAT to use asynchronous operation mode so that the host CPU can context-switch to another process after submitting all tasks.

Compression algorithms: We evaluate multiple compression algorithms that are commonly used in production systems, including Deflate (with static and dynamic Huffman coding), LZ4, and Zstd. IAA only supports Deflate, while QAT supports Deflate, LZ4, and Zstd (compression only). While configuring to a higher compression level offers a higher compression ratio, it comes at the cost of slower performance. Since this paper aims at minimizing the de-/compression overhead, we configure the compression level to the lowest possible (i.e., 1) in all experiments.

Dataset: We use the TPC-H dataset with a scale factor of 10. We use the largest table (`lineitem`) for the characterization experiments. Absolute compression ratios are data-dependent, while the block-size, submission, and saturation trends are the primary focus of our evaluation.

We report throughput in terms of logical uncompressed bytes processed per second for compression and decompression. Each block size denotes the independent compression unit submitted to the CPU library or accelerator API. The 1-thread CPU baseline estimates per-core software efficiency, the 96-thread CPU baseline estimates full-socket software capacity, and the single-device IAA/QAT baselines estimate the capacity of one offload endpoint. This comparison is intended to guide resource allocation rather than to claim that one accelerator replaces an entire socket.

3.2 Block-size sensitivity

As discussed in Section 2.1, compression operates at the granularity of block sizes. We configure and characterize QAT and IAA with different compression block sizes, ranging from 4KB to 2MB. For each block size, we compare four execution configurations: *1 CPU*, a single software thread; *96 CPU*, software execution using all physical cores of the socket; *1 QAT*, one QAT hardware device; and *1 IAA*, one IAA hardware device. Figures 2 to 5 report the resulting de-/compression speed across Deflate dynamic, Deflate static, LZ4, and Zstd (only compression speed as explained in Section 3.4). We use these four configurations because the goal of this paper is not to benchmark codecs in isolation, but to quantify when on-chip acceleration actually removes the compression tax from analytical query execution.

For small block sizes (e.g., 4KB), our measurements show a severe performance collapse for both the 96-thread CPU and a single QAT device. For the 96-thread CPU, this degradation is driven by thread-synchronization and task-scheduling overheads overshadowing compute time. For QAT, the overhead of IOMMU page-table

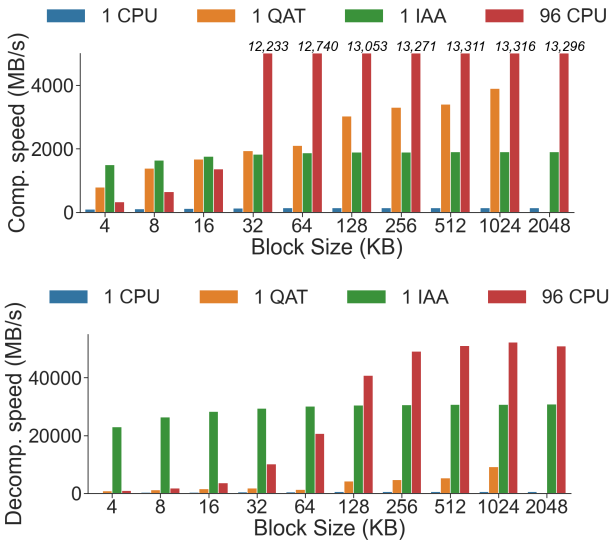

Figure 2: Deflate (dynamic): varying data block size

mapping and background ring buffer polling neutralizes the silicon's speedup for small payloads. In contrast, a single IAA achieves a decompression speed of 23.0 GB/s on 4KB blocks with Deflate (dynamic).

For large block sizes (e.g., 1MB), different hardware accelerators are preferred depending on the compression algorithm. IAA delivers ~30 GB/s decompression speed for Deflate, while QAT caps at ~10 GB/s. Deflate (static) exhibits similar compression performance for both IAA and QAT, while QAT achieves 2× compression throughput for Deflate (dynamic) compared to IAA. Only QAT supports non-Deflate compression algorithms, such as LZ4 and Zstd, and larger blocks improve throughput-oriented QAT offload.

The performance for QAT using a 2MB block size is not included. This is because QATzip restricts the maximum stream buffer size to $(2MB - 5KB)$, though it can be recompiled to support a larger block size. QAT-ZSTD-PLUGIN caps the compression history window at 64KB, and any block size larger than that will be partitioned into independent 64KB data chunks for processing. Therefore, block sizes larger than that are omitted from Figure 5.

> Insight 1: Block size changes the preferred execution path. IAA sustains high de-/compression throughput even at small block sizes, while QAT needs larger blocks to amortize its API-level costs. A full CPU socket remains highly competitive for large-block compression, but consumes cores that could be used for other useful work.

3.3 Accelerator saturation

Offloading compression to on-chip accelerators does not eliminate the compression tax automatically; it shifts the bottleneck from CPU cores to accelerator submission queues, work queues, and device-side scheduling. Treating IAA and QAT as infinitely scalable resources can therefore create queue contention and silent pipeline stalls. To characterize this boundary, we fix the accelerator device

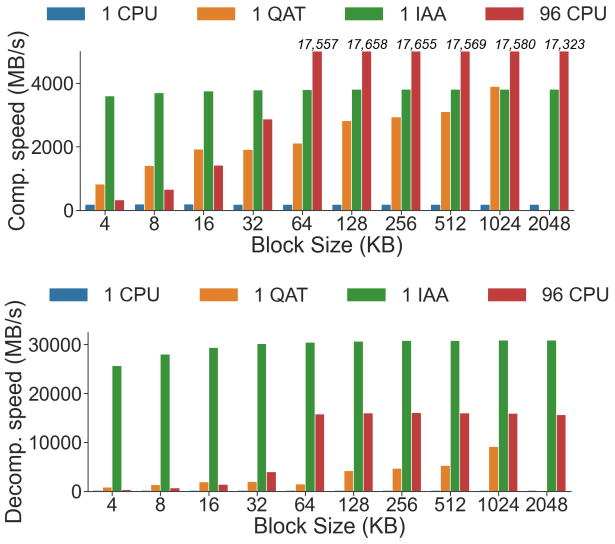

Figure 3: Deflate (static): varying data block size

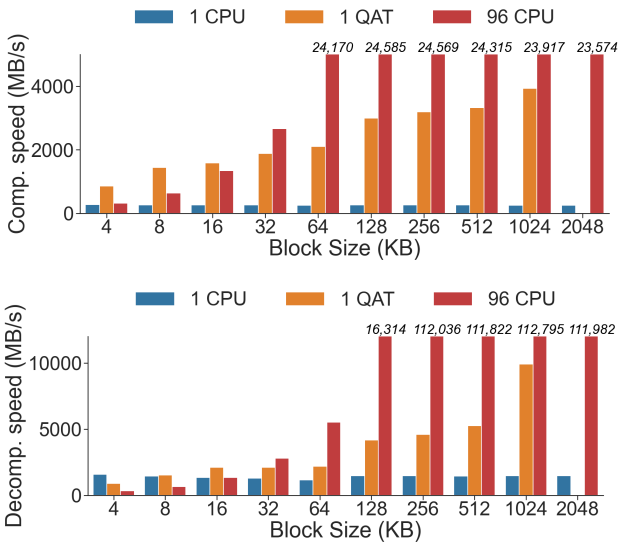

Figure 4: LZ4: varying data block size

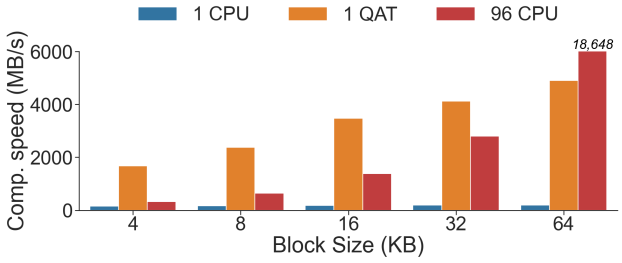

Figure 5: ZSTD compression: varying data block size

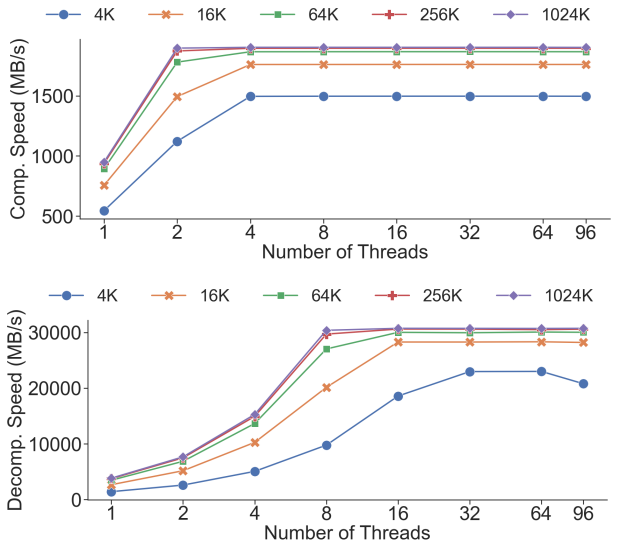

Figure 6: IAA: varying the number of submission threads with Deflate (dynamic)

and vary the number of concurrent submission threads, exposing the architectural limits that determine how analytical engines can offload de-/compression effectively. Figures 6 to 8 show the resulting throughput as submission concurrency increases.

IAA's hardware Work Queues (WQs) have finite depth and over-subscribing does not offer additional de-/compression performance improvement. For the IAA (Figure 6), peak throughput is reached around 4–16 submitters depending on block size and direction.

QAT also has a limited number of ring buffer pairs. As shown in Figures 7 to 8, single QAT performance peaks at 16 submission threads. Pushing concurrency beyond this point actively degrades de-/compression throughput as internal hardware queue contention masks the efficiency of the accelerator.

> Insight 2: On-chip accelerators are finite shared resources. A DBMS should cap per-device submission concurrency and route overflow work to another local device or use software de-/compression rather than blindly increasing the number of submitters.

3.4 Choosing the optimal accelerator / compression algorithm

Given the performance limitations of IAA and QAT discussed above, database engines must dynamically route de-/compression based on specific compression algorithms and available system capacity.

Figures 9 to 11 summarize the speed-ratio tradeoff across the compression formats evaluated in this section. Each point corresponds to one block size under a specific execution configuration: *1 CPU*, *96 CPU*, *1 IAA*, or *1 QAT*. The x-axis reports the achieved compression ratio, while the y-axis reports either compression or

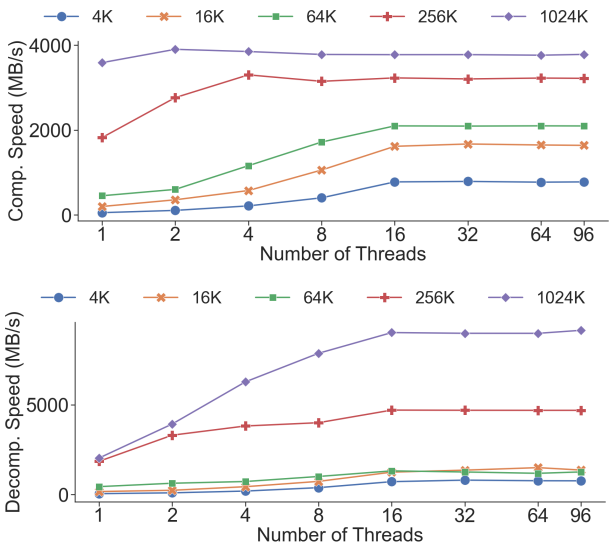

Figure 7: QAT: varying the number of submission threads with Deflate (dynamic)

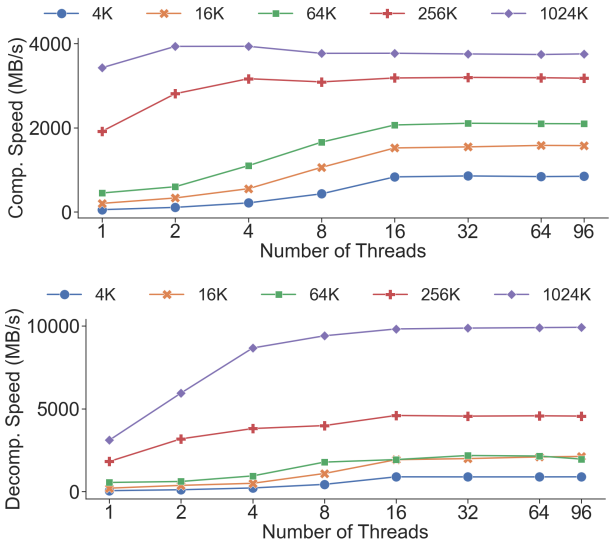

Figure 8: QAT: varying the number of submission threads with LZ4

decompression throughput. Points further to the right provide better space reduction, whereas points higher on the y-axis provide higher throughput.

The plots expose the main codec tradeoffs. First, on IAA, Deflate with dynamic Huffman coding provides the strongest space reduction over Deflate with static Huffman coding: it improves the compression ratio by 25-28% over Deflate with static Huffman coding. This gain comes at a cost: dynamic Deflate is 2–2.4× slower to compress, but it loses at most 10% decompression throughput. On QAT, however, static and dynamic Deflate exhibit very similar compression and decompression throughput, while dynamic Deflate still preserves its 25% compression-ratio advantage. This indicates that QAT hardware offloading makes the extra entropy-coding complexity of dynamic Deflate almost free from a throughput perspective.

Second, on QAT, LZ4 reaches de-/compression throughput comparable to Deflate on the hardware path, but with a 35-40% lower compression ratio. This indicates that once Huffman coding is accelerated efficiently in hardware, Deflate no longer pays the traditional throughput penalty that motivates using general-purpose CPU cores for LZ4. LZ4, therefore, remains attractive for compatibility and simple, fast movement, but its space-efficiency loss is harder to justify when Deflate acceleration is available.

We omit a full Zstd speed-ratio analysis because the current QAT-ZSTD software stack exposes hardware-accelerated compression but does not provide the corresponding decompression path. Nevertheless, the compression results are promising: Zstd reaches a compression ratio close to Deflate with dynamic Huffman coding while providing 2× faster compression speed.

The choice of compression format dictates the offloading path. IAA exclusively supports Deflate and is the optimal choice for fine-to-medium block sizes, prioritizing extreme decompression speed.

If block sizes are small and IAA is saturated (or the format is non-Deflate), general-purpose CPUs must be leveraged as the fallback to avoid QAT's degraded performance with small block sizes.

To achieve the absolute highest decompression bandwidth for massive analytical scans, systems should avoid artificially partitioning tasks exclusively to a single hardware accelerator. By saturating multiple accelerator devices and simultaneously dispatching the remainder to the CPU pool, a system achieves hybrid co-execution with maximum decompression throughput. Our characterization experiment shows that combining the peak decompression performance from the CPU pool (52.3 GB/s) and from four local IAA devices (30.8 GB/s per device) yields an aggregate decompression throughput of 166.2 GB/s.

> Insight 3: Hardware offloading depends on the compression format and block size. IAA is the strongest evaluated hardware path for Deflate, while QAT additionally supports LZ4 and Zstd (compression only), which require large block sizes to maintain efficiency.

> Insight 4: Peak decompression throughput requires hybrid co-execution rather than mutually exclusive task partitioning through concurrently saturating the local accelerators and dispatching the residual workload to the CPU pool.

4 DESIGN IMPLICATIONS FOR A HARDWARE-AWARE SCHEDULER

Section 3 shows that the best execution path depends on the codec, the operation direction, and the block size, so a system cannot choose a path from advertised device bandwidth alone. Choosing from bandwidth treats each accelerator as zero-cost and infinitely scalable, and our measurements refute both assumptions: every

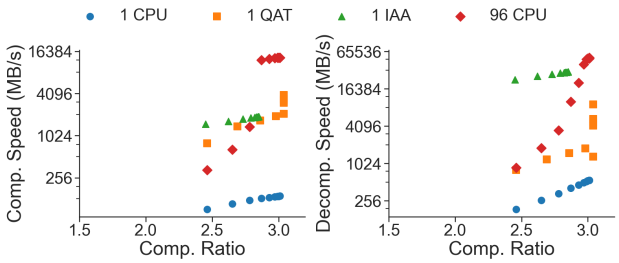

Figure 9: Deflate (dynamic): De-/Comp. speed over comp. ratio.

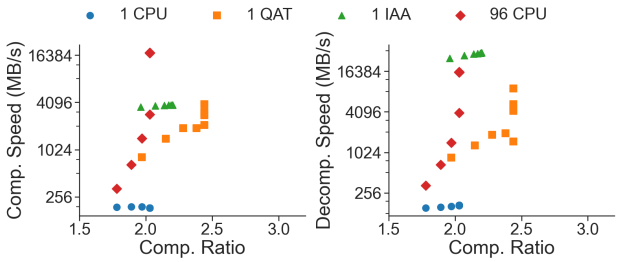

Figure 10: Deflate (static): De-/Comp. speed over comp. ratio.

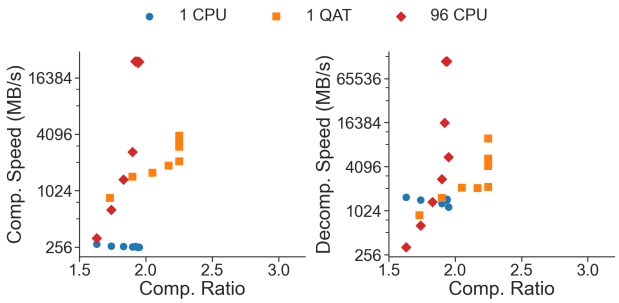

Figure 11: LZ4: De-/Comp. speed over comp. ratio.

offload pays a submission cost that dominates latency at small block sizes, and a single device saturates at modest concurrency. A hardware-aware scheduler must account for both limits when it routes a block to IAA, QAT, or the CPU.

This section quantifies the inputs such a scheduler needs. Section 4.1 measures offloading overhead and the block size at which it is amortized. Section 4.2 translates the per-device saturation limit (Section 3.3) into a concurrency cap and an overflow policy. Section 4.3 bounds the aggregate throughput available when de-/compression is spread across the CPU and all local accelerators rather than pinned to one path. We frame these as design implications for a hardware-aware scheduler and leave a complete implementation to future work.

4.1 Cost modeling of offloading overhead

To make dynamic offloading decisions, the scheduler must be able to estimate the latency for decompressing a data block of uncompressed size S using the CPU pool (T_{CPU}) versus the hardware accelerator (T_{accel}). Similar to established cost modeling for accelerators [11], we partition the overall execution time into a constant dispatch overhead and the hardware execution phase. We model the end-to-end latency of an offloaded de-/compression task as:

$$T_{accel} = T_{submit} + T_{HW\_execution} + T_{poll}$$

As indicated by T_{accel}, the offloading lifecycle is composed of three phases: submission (the time spent formatting the descriptor and issuing the `ENQCMD` instruction), de-/compression execution on the hardware, and polling (the host's wait for the hardware to write the completion record to memory).

The active hardware execution time is dictated by the asymptotic peak throughput of the silicon ($Xput_{accel\_peak}$), which can be bottlenecked by either the internal accelerator engines or the fabric bandwidth. The submission and polling phases represent the overhead (O) that must be paid to move offload tasks across the system bus, including the instruction dispatch latency, the software stack overhead, memory access overhead, and the internal silicon setup. Therefore, we formulate the hardware execution time as:

$$O(S) = T_{submit} + T_{poll}$$

$$T_{accel}(S) = O(S) + \frac{S}{Xput_{accel\_peak}}$$

We design a microbenchmark to isolate and measure the offloading overhead as block sizes vary from 4 KB to 1024 KB. We use the `__rdtsc()` hardware intrinsic to count CPU cycles, then convert them to microseconds given the locked CPU frequency at 2.1 GHz. IAA allows user-space applications to submit work descriptors directly to a shared work queue using the `ENQCMD` instruction. This enables us to split the timer. We measure T_{submit} around the asynchronous `qpl_submit_job` API call (capturing parameter validation and descriptor dispatch), and T_{poll} around the `qpl_wait_job` call (capturing the hardware execution and completion write). QAT submits requests through descriptor rings in host memory, incurring MMIO tail updates upon publication. We configure QAT to use asynchronous operation mode so that the host CPU can context-switch after submitting tasks. Because the QATzip library abstracts submission, doorbell ringing, and completion polling into background threads, we wrap the entire asynchronous API invocation to measure the total end-to-end round-trip time ($T_{submit} + T_{poll}$).

As listed in Table 2, IAA demonstrates a highly consistent submission latency, averaging approximately 0.65 μs, independent of payload size. This 0.65 μs latency represents the irreducible software abstraction overhead required by the Intel QPL library to validate parameters and format the `qpl_job` descriptor before dispatch. While minimal, it must be explicitly accounted for in the scheduling logic.

IAA establishes a minimum offloading overhead of 4.2 μs for small block sizes (4 KB), while QAT floors at 69.3 μs. This overhead stems from QAT's MMIO tail update and its reliance on background threads polling the completion rings, a cost paid even for zero-byte-equivalent workloads.

	IAA		QAT
Block Size	T_{submit} (μs)	T_{poll} (μs)	$T_{submit} + T_{poll}$ (μs)
4 KB	0.65	3.53	69.30
8 KB	0.65	6.00	71.02
16 KB	0.64	9.96	70.84
32 KB	0.64	18.33	83.04
64 KB	0.65	34.50	124.38
128 KB	0.66	67.38	91.49
256 KB	0.67	133.00	152.34
512 KB	0.66	264.30	184.60
1 MB	0.67	526.08	328.29

Table 2: IAA and QAT's measured offloading overhead across varying block sizes. The latency inversion observed in QAT between 64 KB and 128 KB likely stems from QATzip chunking payloads at 64 KB (`QZ_HW_BUFF_SZ`).

Scaling to larger block sizes, QAT's offloading overhead amortizes across batched requests, whereas IAA scales linearly. This creates a compression throughput crossover point and validates our insights from Section 3.2: QAT is heavily penalized at small granularities, while IAA sustains high throughput across block sizes.

4.2 Bypassing the hardware traps

Past its saturation point, an accelerator is a scheduling hazard rather than spare capacity, because additional submissions degrade throughput instead of improving it (Section 3.3). In shared work queue (SWQ) mode, the CPU submits jobs by asynchronous spinning, so a scheduler must track per-device queue depth and, once a device reaches its 16-thread saturation limit, divert overflow blocks to an idle device or to the CPU rather than oversubscribing.

4.3 Peak system capacity via hybrid co-execution

When optimizing high-throughput analytical pipelines, the ultimate objective is not to choose between hardware and software, but to maximize combined system throughput through hybrid co-execution. By applying the routing rules derived in Section 3, a system can saturate all available accelerators up to their physical limits and assign the remainder to the CPU.

In our single-socket configuration, combining these peak decompression speeds for Deflate dynamic yields 52.3 GB/s from the CPU pool, 30.8 GB/s from a single IAA device, and 9.1 GB/s from a single QAT device. Given that there are a total of 4 IAA and 4 QAT devices per socket, an ideal hybrid co-execution with all available resources and perfect load balancing would deliver a theoretical aggregate decompression throughput of 212 GB/s.

5 CONCLUSION

This paper characterizes how Intel IAA, Intel QAT, and CPU software behave for database-relevant compression granularities. The results show that on-chip acceleration can reduce the CPU cost of compression and decompression, but only when the DBMS chooses the execution path with awareness of codec support, block size, operation direction, and accelerator saturation.

IAA is the strongest evaluated hardware path for Deflate decompression and remains effective on small blocks. QAT supports a broader set of formats, including LZ4 and Zstd compression, but requires sufficiently large blocks or batching to amortize its higher API-level costs. A full CPU socket remains highly competitive for several large-block compression cases, so hardware offload should be viewed as a resource-allocation decision rather than a universal replacement for software.

The main scheduling implication is that analytical engines should route compression work dynamically: use local accelerators up to their saturation point, avoid cross-socket offload unless measured, and fall back to CPU software when codec support, block size, or queue state makes hardware inefficient. This hybrid view is the path toward reducing the compression tax without overloading either CPU cores or accelerator queues.

While our characterization isolates accelerator performance on de-/compression, hardware offloading must ultimately be evaluated within the context of a complete query engine. Future work will focus on integrating our proposed dynamic scheduler into an end-to-end analytics engine. This is necessary to evaluate the interplay between on-chip general-purpose compression and database-specific lightweight encodings [22, 34] (e.g., dictionary coding [17, 20], run-length encoding, frame-of-reference, bit-packing [10, 44], and learned schemes [19, 26, 37]) to determine their combined efficacy in real-world storage formats. It will allow us to assess how hardware offloading interacts with core database operators (e.g., scans, predicate evaluation, joins, and aggregations) and complex system dynamics, including cache pollution, memory bandwidth contention, NUMA effects, and concurrent query execution.

ACKNOWLEDGMENTS

This work was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) for the Prodasys project, a European Research Council (ERC) Advanced Grant.

REFERENCES

[1] 2019. Data Compression in PostgreSQL and Its Future. https://www.postgresql.eu/events/pgconfeu2019/sessions/session/2671/slides/263/Data_Compression_in_PostgreSQL_and_its_future_noscript.pdf PGConf.EU 2019 slides.
[2] 2025. lzbench: In-memory benchmark of open-source LZ77/LZSS/LZMA compressors. https://github.com/inikep/lzbench
[3] 2026. Fluent Bit Amazon S3 output. https://docs.fluentbit.io/manual/data-pipeline/outputs/s3.
[4] 2026. InnoDB page compression. https://dev.mysql.com/doc/refman/8.4/en/innodb-table-compression.html.
[5] 2026. Intel® Query Processing Library (Intel® QPL). https://github.com/intel/qpl.
[6] 2026. Intel® QuickAssist Technology Library (QATlib). https://github.com/intel/qatlib.
[7] 2026. Intel® QuickAssist Technology (QAT) QATzip Library. https://github.com/intel/qatzip.
[8] 2026. Intel® QuickAssist Technology ZSTD Plugin. https://github.com/intel/QAT-ZSTD-Plugin.
[9] 2026. LZ4: Extremely fast compression. https://lz4.org/.
[10] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548
[11] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th*

Annual International Symposium on Computer Architecture (ISCA). 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[12] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A General-Purpose Data Compressor. *ACM Trans. Inf. Syst.* 37, 1, Article 4 (Dec. 2018), 30 pages. https://doi.org/10.1145/3231935

[13] Amazon Web Services. 2026. *Amazon Simple Storage Service (Amazon S3)*. Amazon.com, Inc. https://aws.amazon.com/s3/

[14] Apache Software Foundation. 2023. KAFKA-14629: Performance improvement for Zstd compressed workload. https://issues.apache.org/jira/browse/KAFKA-14629

[15] Alexander Baumstark, Laurin Martins, and Kai-Uwe Sattler. 2025. Uncore your Queries: Towards CPU-less Query Processing. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/3736227.3736243

[16] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054

[17] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 283–296. https://doi.org/10.1145/1559845.1559877

[18] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698812

[19] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A Learned Approach to Design Compressed Rank/Select Data Structures. *ACM Trans. Algorithms* 18, 3, Article 24 (Oct. 2022), 28 pages. https://doi.org/10.1145/3524060

[20] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proc. VLDB Endow.* 13, 12 (July 2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[21] ClickHouse. 2026. Database compression: encodings, codecs and ratios | Engineering | ClickHouse Resource Hub. https://clickhouse.com/resources/engineering/database-compression

[22] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3, Article 9 (June 2019), 46 pages. https://doi.org/10.1145/3323991

[23] P. Deutsch. 1996. RFC1951: DEFLATE Compressed Data Format Specification version 1.3.

[24] Jarek Duda. 2013. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR* abs/1311.2540 (2013). arXiv:1311.2540 http://arxiv.org/abs/1311.2540

[25] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 47, 16 pages. https://doi.org/10.1145/3579371.3589082

[26] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned Compression of Nonlinear Time Series with Random Access . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1579–1592. https://doi.org/10.1109/ICDE65448.2025.00122

[27] Qingda Hu, Xinjun (Jimmy) Yang, Feifei Li, Junru Li, Ya Lin, Yuqi Zhou, Yicong Zhu, Junwei Zhang, Rongbiao Xie, Ling Zhou, Bin Wu, and Wenchao Zhou. 2026. PolarStore: High-Performance Data Compression for Large-Scale Cloud-Native Databases. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association, Santa Clara, CA, 509–525. https://www.usenix.org/conference/fast26/presentation/hu

[28] Intel Corporation. 2025. *Programmer's Guide: Intel® QuickAssist Technology, Hardware Version 2.0*. Intel Corporation. https://www.intel.com/content/www/us/en/content-details/843378/intel-quickassist-technology-intel-qat-programmer-s-guide-for-linux-for-hardware-version-2-0.html

[29] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. 2023. STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 619–633. https://www.usenix.org/conference/atc23/presentation/ji

[30] Muhammad Attahir Jibril, Hani Al-Sayeh, and Kai-Uwe Sattler. 2024. Accelerating Aggregation Using a Real Processing-in-Memory System. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3920–3932. https://doi.org/10.1109/ICDE60146.2024.00300

[31] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[32] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2, Article 118 (June 2023), 26 pages. https://doi.org/10.1145/3589263

[33] Christos Laspias, Andrew Pavlo, and Jignesh M. Patel. 2025. A Hot Take on the Intel Analytics Accelerator for Database Management Systems. In *ADMS@VLDB*. https://db.cs.cmu.edu/papers/2025/laspias-adms2025.pdf

[34] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. https://doi.org/10.1002/spe.2203 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2203

[35] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[36] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (July 2023), 3044–3056. https://doi.org/10.14778/3611479.3611507

[37] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (March 2024), 28 pages. https://doi.org/10.1145/3639320

[38] Zhiheng Liu and Shaoxu Song. 2026. Improving LZ4 for Effective Compression and Efficient Query. *Proc. ACM Manag. Data* 4, 1, Article 46 (April 2026), 26 pages. https://doi.org/10.1145/3786660

[39] Alistair Moffat. 2019. Huffman Coding. *ACM Comput. Surv.* 52, 4, Article 85 (Aug. 2019), 35 pages. https://doi.org/10.1145/3342555

[40] Praveen Mosur. 2024. Built for the Edge: The Intel® Xeon® 6 SoC. In *2024 IEEE Hot Chips 36 Symposium (HCS)*. 1–28. https://doi.org/10.1109/HCS61935.2024.10665220

[41] Hamish Nicholson, Konstantinos Chasialis, Antonio Boffa, and Anastasia Ailamaki. 2025. The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/3736227.3736240

[42] Keyur Panchal. 2024. Optimizing PostgreSQL Performance & Compression: pglz vs. LZ4. https://www.tigerdata.com/blog/optimizing-postgresql-performance-compression-pglz-vs-lz4

[43] Yiming Qiao, Yihan Gao, and Huanchen Zhang. 2024. Blitzcrank: Fast Semantic Compression for In-Memory Online Transaction Processing. *Proc. VLDB Endow.* 17, 10 (June 2024), 2528–2540. https://doi.org/10.14778/3675034.3675044

[44] Mark A. Roth and Scott J. Van Horn. 1993. Database compression. *SIGMOD Rec.* 22, 3 (Sept. 1993), 31–39. https://doi.org/10.1145/163090.163096

[45] The DuckDB team. 2025. Announcing DuckDB 1.2.0. https://duckdb.org/2025/02/05/announcing-duckdb-120#zstd-compression

[46] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2, Article 195 (June 2023), 27 pages. https://doi.org/10.1145/3589775

[47] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, Ipoom Jeong, and Nam Sung Kim. 2024. Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 848–862. https://doi.org/10.1109/ISCA59077.2024.00066

[48] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 148–161. https://doi.org/10.14778/3626292.3626298

[49] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[50] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934

[51] Zstandard Project. 2017. Zstandard: Fast real-time compression algorithm. https://facebook.github.io/zstd/