A Hot Take on the Intel Analytics Accelerator for Database Management Systems

Christos Laspias
Carnegie Mellon University
claspias@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

ABSTRACT

For as long as database management systems (DBMSs) have existed, there have been efforts to develop specialized hardware to accelerate their workloads. The goal is clear: to offload the DBMS's most common and repetitive tasks to hardware, thereby improving efficiency and performance. Recently, Intel has released CPUs with new accelerators located on the same die, such as the In-Memory Analytics Accelerator (IAA) that targets data processing tasks. In this work, we examine the Intel IAA's ability to optimize data compression and decompression operations for online analytical processing (OLAP) workloads. To evaluate the benefits of this accelerator, we added support for IAA compression into DuckDB. Our experiments comparing IAA with DuckDB's existing compression method (Snappy) show that it improves decompression speeds by up to 10× in microbenchmarks and the end-to-end TPC-H query latencies by up to 30%.

VLDB Workshop Reference Format:
Christos Laspias, Andrew Pavlo, and Jignesh M. Patel. A Hot Take on the Intel Analytics Accelerator for Database Management Systems. VLDB 2025 Workshop: 16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS25).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ChrisLaspias/iaa-adms25.

1 INTRODUCTION

Hardware advances often pave the way for higher-performance DBMSs, aiding both in lower latency and higher bandwidth transactions and analytical workloads. CPUs have evolved tremendously over the past 20 years, incorporating higher core counts, simultaneous multithreading (SMT), increased clock frequencies, and both larger and more caches. In addition to improved processor performance, the low price per byte of main memory enabled significant growth in their capacity. As a result, system designers began building in-memory DBMSs for workloads that could entirely reside in RAM [20].

The trend continued for many years, motivating the creation of many high-performance in-memory OLTP and OLAP systems [10, 20, 21, 23], where compression was less critical for performance than in a disk-based system. However, in recent years, the size of the workloads has outpaced the growth in main memory capacity, hindering systems from relying on purely in-memory processing. Instead, newer single-node systems employ a disk-based approach [28], whereas cloud systems are built in a shared-nothing architecture. Compression is particularly important for those systems, trading off CPU cycles for (i) reduced storage and therefore less I/Os and (ii) faster data transmission over the network. However, the tremendous improvements in high-bandwidth SSDs and low-latency networks, along with the single-core CPU performance stagnation due to the slowing of Moore's law, make many single-threaded compression algorithms the bottleneck in those settings. Moreover, compression is paramount due to the shift to the cloud and the widespread adoption of open-source file formats such as Parquet [5], which heavily rely on compression. The aforementioned reasons make hardware accelerators for compression relevant to DBMSs since hardware-assisted compression can overcome CPU bottlenecks.

An effective compression algorithm balances compression speed, compression ratio, and decompression speed. Thus, the choice of algorithm can notably affect system performance based on workload and resource constraints. The Parquet file format, for example, supports various compression algorithms, including zstd [11], Snappy [15], and LZ4 [14]. These "heavyweight" algorithms provide different tradeoffs between compression ratio and (de)compression speeds.

Although some compression algorithms have evolved and become more amenable to parallelism, many are inherently difficult to parallelize and rely on single-core performance (e.g. Snappy). To address this, Intel recently introduced the Intel In-Memory Analytics Accelerator (IAA) in the 4th generation Xeon Scalable processors, designed to accelerate analytical operations and boost the performance of analytical engines. In this paper, we integrate Intel IAA in DuckDB [30] and examine compression and decompression, since these operations account for a significant portion of the execution time for TPC-H when reading from Parquet files. Our results indicate that IAA offers the fastest compression and decompression speed in most scenarios, with a competitive compression ratio. When compressing the TPC-H dataset, IAA outperforms Snappy, the default compression algorithm for Parquet in DuckDB, sometimes achieving an order of magnitude faster compression and decompression.

The remainder of this paper is organized as follows: Section 2 contains background information on the Parquet file format. Section 3 contains information Intel's IAA. In section 4 we describe the implementation of the Intel IAA (de)compression in DuckDB. Section 5 contains our experimental evaluation. We discuss related work in section 6. Section 7 contains our concluding remarks.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

2 BACKGROUND

2.1 The Parquet file format

Parquet uses a columnar data representation, which is inspired by the PAX [4] layout. Parquet first partitions a table horizontally, splitting it into *row groups*. Each row group holds a subset of the total rows. For example, for a table with 1M rows, if we choose to use two row groups, then the first row group would include rows [0, 499,999] and the second row group would include rows [500,000, 999,999]. Within each row group, every column of the table is stored sequentially in a *column chunk*. Finally, every column chunk stores the values of one column in multiple pages. Figure 1 depicts a simplified version of the Parquet file format, without metadata, footer, zone maps, etc.

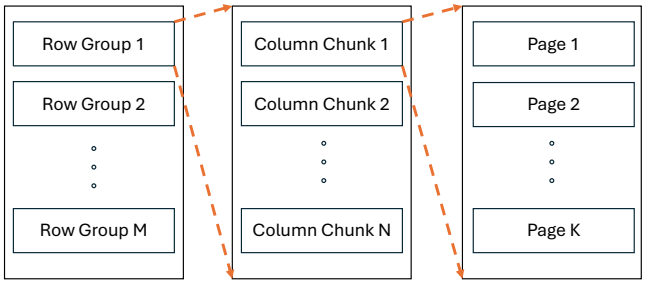

Figure 1: Simplified Representation of the Parquet File Format

2.2 Compression in Parquet

Parquet employs compression to reduce the size of the data, since compression reduces both network and I/O costs [1]. Parquet was designed for big-data processing and thus utilizes a columnar storage format. Exploiting this design, Parquet offers various lightweight compression schemes (encodings), since with columnar representation, values of the same type are stored together and can be naturally compressed efficiently with simple encodings (e.g. run length encoding). Parquet first compresses pages with dictionary encoding by default, and on top of that, it either applies run length encoding (RLE) or bitpacking to the dictionary codes [34] . After encoding the values of a page with Dictionary Encoding + {RLE or Bitpacking}, Parquet allows for block compression using one of the following algorithms: Uncompressed, Snappy, GZIP, LZO, BROTLI, LZ4, and zstd. The granularity of block compression is a page within a column chunk within a row group. This is an essential detail since pages are by default set to 1MB, and some algorithms work better/worse for larger/smaller blocks. Moreover, since some block compression algorithms (e.g., Snappy) are single-threaded, splitting the data into multiple independent pages enables parallel compression, with each thread handling a separate page. Each block compression algorithm offers different tradeoffs. For example, zstd offers the best compression ratio for the TPC-H `lineitem` table with reasonable decompression speed when set to the lowest compression level. Snappy strikes a good balance between a reasonable compression ratio and fast decompression speed. Therefore, choosing a compression algorithm is essential to the performance of end-to-end query latencies.

The lightweight encodings including RLE, Delta encoding, bitpacking, and dictionary encoding are particularly effective for compressing columnar data, which often exhibit patterns such as repeated values, small value ranges, or minimal differences between consecutive entries. Recent efforts [22] have explored cascading these lightweight encodings, achieving competitive compression ratios with very fast (de)compression speeds. Despite their effectiveness and efficiency, many DBMSs still apply "heavyweight" compression on top of them for additional storage savings.

3 INTEL IAA

In this section, we describe the Intel In-Memory Analytics Accelerator (IAA). Datacenter tax has been recently characterized as CPU cycles that are spent on low-level operations, including memory moves, compression, and encryption [13]. These operations take a significant portion of the CPU cycles spent on applications running in the cloud. The hardware community has tried to respond to this challenge by designing specialized hardware, like FPGAs, which are better for handling specific compute-intensive tasks but lack generality. FPGAs have been used in the cloud to accelerate various operations, with video encoding being a prominent example. However, with hardware specialization, there is additional complexity for the system developer as programming such devices requires more effort than using a more familiar toolchain for CPU programming (compiler, OS, profiler, debugger). Intel IAA aims to improve the performance of common analytical operations, with a simple developer experience.

3.1 Memory Management and Task Submission

Intel IAA is a built-in accelerator residing within the CPU's die, designed to accelerate common analytical operations in response to the increasingly high volumes of data that need to be processed in datacenter environments, such as cloud data warehouses. As an in-die accelerator, Intel IAA operates alongside the CPU and, therefore, can utilize existing hardware structures present in modern CPUs. Specifically, Intel IAA is able to work closely with the CPU's cache infrastructure. IAA enables data placement into the main memory or directly in lower-level cache (LLC) after an operation, reducing latency and data movement overhead. For example after a decompression task, data can be placed directly into the LLC.

Memory management is challenging when interacting with heterogeneous hardware. Unlike discrete accelerators (e.g., GPUs), that require communication over the PCIe bus, IAA leverages the CPU's memory subsystem directly. IAA's design eliminates the need for explicit memory transfers and exposes a transparent programming model to developers. IAA shares the virtual address space with the CPU via Shared Virtual Memory (SVM). Whenever IAA accesses a virtual address, that requires translation, IAA queries the Address Translation Cache (the "TLB" for IAA); if the translation is not cached, then IAA fetches the translation through the Input-Output Memory Management Unit (IOMMU). On a page fault, the IOMMU handles the fault resolution similarly to the CPU, ensuring consistent memory access behavior [33].

SVM has two main benefits for DBMSs when using the IAA. First, since the accelerator uses the CPU's memory subsystem and shares

the same virtual address space, it can handle large datasets, avoiding expensive data movements from and to the accelerator. Second, as described later, submitting a task to the accelerator is easier and faster using Memory Mapped I/O instead of going through the OS kernel since the accelerator can use the CPU's memory subsystem. Submitting a task to the accelerator (e.g., compression) involves writing a job description in a memory-mapped region and notifying the accelerator to read it and execute it. To efficiently submit a task to the accelerator and ensure atomicity, Intel introduced new direct-store instructions (MOVDIRI/MOVDIR64B and ENQCMD) that bypass the cache hierarchy and write the job description in the memory-mapped region. The job description is 64 bytes long (equal to the size of a cache line in modern processors). All the necessary information for a task is placed in those 64 bytes, such as which operation to execute (e.g., compression and filter), where to read the data from, and where to write the result.

3.2 Hardware Structure

Intel IAA consists of Acceleration Engines (AE) and Work Queues (WQ). The AEs are the execution units responsible for executing a task. The WQs are buffers that store task descriptions for the AEs to execute when the AEs are not occupied. WQ can be configured by users in shared mode (SWQ) where multiple clients (threads, processes, etc) can submit tasks to the WQ. In addition, WQ can also be configured by users in dedicated mode (DWQ), allowing only one client to submit a task to the WQ. In the DWQ, the MOVDIRI/MOVDIR64B instructions can place a task since no synchronization is needed. In the case of SWQ, since multiple clients might be submitting jobs in the SWQ, the newly introduced ENQCMD is used (in the Query Processing Library [19] described later), which also notifies the programmer whether the task has been successfully placed in the SWQ by modifying the contents of a register. Users can configure the WQs by using the accel-config library [18], which allows switching the modes of WQs and other settings for the accelerator. The AEs are responsible for executing tasks (e.g., compression). This model is particularly effective for parallelized compression scenarios—such as when large files are split into independent pages—because multiple tasks can be enqueued and processed concurrently. The newly introduced instructions (e.g., ENQCMD) allow for task placement without the need for expensive locking.

3.3 Operations

We describe the operations offered by IAA. For the scope of this paper, however, we only experiment with one category of operations, namely the compression and decompression operations as this functionality in the IAA offered the most promising benefits for analytic database operations. IAA also offers hardware-accelerated CRC calculations, though we found this feature less applicable to the TPC-H workload.

Intel IAA offers two main execution blocks, *compression* and *analytics*. The *analytics* execution block supports operations such as *decompression*, *filtering*, or a pipelined combination of both. In the pipelined mode, the output of the *decompression* stage is passed to the *filter* stage without requiring separate job submissions, thereby reducing overhead.

IAA supports *compression* via the Deflate algorithm [9]. Deflate is a lossless compression algorithm based on LZ77 [35] and Huffman coding [17]. LZ77 uses a sliding window while compressing the data and replaces repeated sequences of bytes called "strings" with a reference to their earlier occurrence. Therefore, only one copy of each "string" is maintained while subsequent occurrences are replaced with a pair of distance-length (the next length characters are the same as distance characters before those). Huffman coding seeks to minimize the size of the data by assigning codes to each character. The assignment of the codes matters, and the algorithm replaces higher frequency characters with short codes, while lower frequency characters with longer codes. In order to find the codes to be assigned to the characters, a Huffman tree is computed. A Huffman tree can be constructed dynamically or statically based on statistics from the data. Constructing a dynamic Huffman tree requires more time but yields higher compression ratios, whereas using a static Huffman tree reduces the construction time before the algorithm starts assigning the codes. IAA offers different options when it comes to choosing the Huffman Tree construction. One can use a precomputed Huffman table (which might be suboptimal), but the codes are "fixed" and are not included in the compressed stream. Another option includes constructing the Huffman Tree dynamically, which involves two passes on the data: one to analyze and gather statistics, and one to compress the data using the "better" Huffman Tree. A final option called the "Canned" mode allows one to provide the algorithm with a static Huffman Table. Deflate is the core compression algorithm used in ZLIB and GZIP. In fact, one can create compatible ZLIB or GZIP formats by writing the necessary ZLIB or GZIP header and footer information. With IAA, this can be achieved by specifying an extra flag in the job description.

The *analytics* functional block performs decompression, filter operations, and CRC computation. We describe the filter operations here. IAA supports four different filter operations: scan, extract, select, and expand.

The scan operation filters values based on a predicate. The output of the scan operation is a bit vector with a bit set to 1 if the value satisfies the predicate, or 0 if the value does not. However, one can specify the bit length of the output vector to be 32 instead of 1, which essentially creates an index vector instead of a bit vector. The scan operation supports the following predicates: $>, \geq, <, \leq, =, \neq, x \geq c1$ AND $x \leq c2, x < c1$ OR $x > c2$.

Similar to BitWeaving/H [25], one can also specify the bit length of the input values to be filtered, allowing IAA to work directly with a bit-packed representation.

The extract operation is relatively simple and, as the name suggests, extracts elements whose indices fall within a specified range (e.g., extract all elements whose indices are between c1 and c2).

The select operation is a more general version of the extract operation. In addition to the input values, the select operation takes as input a bit vector. The operation outputs the elements whose corresponding bit in the bit vector is set to 1 and ignores those whose corresponding bit is set to 0. This operation could be useful when one wants to shrink a vector after applying a selective predicate with the scan operation.

Finally, the expand operation is the opposite of the select operation. For a given bit vector and a vector of elements, expand writes

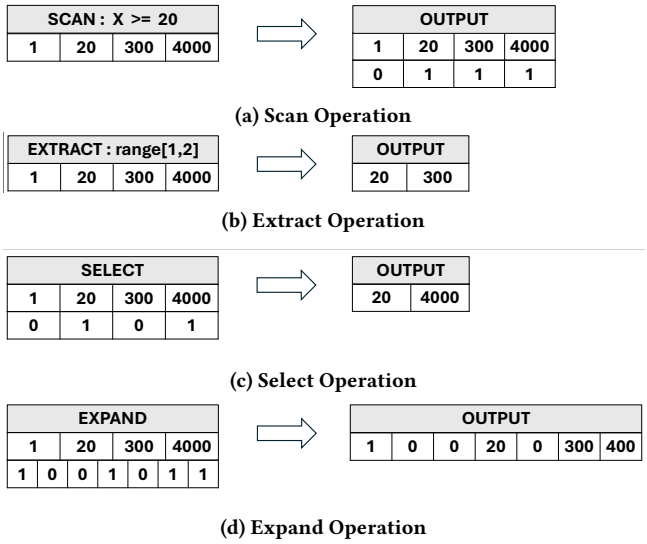

Figure 2: Filter Operations Supported by IAA

a zero to the output if the current bit is set to 0, and writes the next element if the current bit is set to 1.

All operations discussed in this section are illustrated in fig. 2. On the left-hand side, fig. 2 shows the operation to be executed along with the input data (a vector with four elements). On the right-hand side, the output of each operation is shown. The scan operation is shown in fig. 2a. Scanning for values > 20 in a buffer produces a bit vector with each bit set to 1 if the value satisfies the predicate, or 0 if the value does not. Each value is assumed to be 32 bits, but this operation can support arbitrary bit-long numbers, e.g., encoded with 7 bits. The next operation, *extract* (see fig. 2b), as the name suggests, extracts elements that reside inside the specified range. In this case, we extract all the elements from index 1 up to index 2 inclusively. This operation also supports specifying the bit length of the numbers (in this case, it is assumed to be 32 bits). Next, given a bit vector as part of the input, one can use the *select* operation to select only the elements whose corresponding bit in the bit vector is active. In fig. 2c, we select only the elements 20 and 4000. Lastly, given a vector with elements (in this case 4) and a bit vector with the same number of active bits, the *expand* operation writes a 0 to the output if the current bit is 0. Otherwise, it writes the next element from the vector to the output. In fig. 2d, the first bit is set to 1, so the first element in the output is 1. The next two inactive bits follow, so two 0's are written to the output. After that, an active bit follows, and therefore, the next element from the vector, 20, is written to the output. The same pattern continues until we iterate through the entire bit vector.

4 IMPLEMENTATION

To interact with the IAA, Intel provides developers with the open-source library called Intel QPL (Query Processing Library) [19]. Intel QPL allows a developer to specify the execution path of an operation, which is either the software path or the hardware path. The software path executes an operation using optimized code

```
vector<uint8_t> source(uncompressed_size); // Uncompressed buffer
vector<uint8_t> destination(compressed_size); // Compressed buffer

job->op = qpl_op_compress; // Specify compression operation
job->level = qpl_default_level; // Specify compression level

job->next_in_ptr = source.data(); // Uncompressed buffer pointer
job->available_in = source.size(); // Uncompressed size

job->next_out_ptr = destination.data(); // Compressed buffer pointer
job->available_out = destination.size(); // Compressed size

job->flags = QPL_FLAG_FIRST | \ /* flags for compression*/
          QPL_FLAG_LAST | \
          QPL_FLAG_DYNAMIC_HUFFMAN | \
          QPL_FLAG_OMIT_VERIFY;

// Execute the job
qpl_execute_job(job);
```

Figure 3: Simplified Pseudocode for Executing Compression in IAA

provided by Intel. The operations are implemented in modern C++. If the hardware path is chosen, then the computation is offloaded to the IAA. We leverage Intel QPL and integrate it inside DuckDB. We modify DuckDB's Parquet Reader and Writer for compression and decompression to equip DuckDB with an option to use Deflate as a compression algorithm when reading or writing from/to a Parquet file. A 64-byte struct must be filled with the necessary information about the operation to be offloaded to IAA. Figure 3 depicts a simplified version of the code to offload the deflate compression with a dynamic Huffman tree.

The API is simple and can do all the heavy lifting for offloading operations to IAA. The error checking is omitted due to space limitations. IAA supports synchronous and asynchronous execution of operations. Specifically, in synchronous execution, while the IAA is executing the operation, the CPU (thread) blocks and waits for the operation to finish. To achieve asynchronous execution, the operation can be offloaded to the IAA, and the application should periodically check for completion, freeing up the CPU to do other work while the operation is executing in the IAA. For simplicity, we chose synchronous execution for our experiments, but future work will include asynchronous execution. When compression is performed, IAA can also compute a CRC value for the original data and store it alongside the compressed data. During decompression, that CRC value can be used for checksumming. In the pseudocode, we omit the verification to avoid the CRC computation. Decompression is similar to Figure 3 and is omitted.

5 EXPERIMENTAL EVALUATION

We conducted all our experiments in AWS using an m7i.metal-24xl instance using Ubuntu 24.04 LTS with kernel version 6.11. The machine is equipped with a 4th Generation Intel Xeon Scalable Processor (Sapphire Rapids) based on the Golden Cove microarchitecture. Specifically, we used a single socket Intel Xeon Platinum 8488C, which has 96 vCPUs (48 cores, 96 threads). Each core has a 48 KiB private L1 data cache and a private 2 MiB L2 data cache, while a 105 MiB L3 cache is shared among all cores. The machine also has 384 GB of DDR5 ECC RAM, with a clock frequency of 4800 MHz. We also used DuckDB v1.2.2 for all of our experiments, compiled with GCC 13.3.0 -O3. After generating the TPC-H dataset, we used DuckDB's Parquet implementation to write to Parquet files.

Table 1: Compression Metrics of `lineitem` table

Algorithm	Comp. Ratio	Comp (GB/s)	Decomp (GB/s)
Snappy	1.97	0.37	1.12
LZ4	1.97	0.50	2.97
zstd-1	3.02	0.34	1.15
zstd-3	3.12	0.23	1.07
zstd-9	3.38	0.05	1.08
Deflate CPU	**16.21**	2.52	3.27
Deflate IAA	12.77	**4.59**	**18.24**

The DuckDB Parquet reader is also leveraged to read the data from Parquet files during query execution. The CPU is equipped with 4 IAA devices. Each IAA device has 8 AEs, for a total of 32 AEs. For (de)compression, we allocate and submit a new job on the fly for every (de)compression operation. However, a more efficient implementation could create a job pool upon initialization of DuckDB (allocate as many jobs as threads) and reuse the same jobs in every invocation of (de)compression.

5.1 Experimental Results

We profiled TPC-H at scale factor 10 in DuckDB using perf and found that, on average, almost 40% of the CPU cycles are spent on decompressing the data when using Snappy across the whole benchmark. In some queries, decompression accounted for as much as 65% of total CPU cycles. Therefore, compression and decompression are computationally intensive operations that should be accelerated even when using a relatively fast compression algorithm and are highly impactful for query processing. We experimented with different compression algorithms and compared those in a microbenchmark that compresses and decompresses an uncompressed Parquet file that contains the `lineitem` data. In this microbenchmark, we load the Parquet file into an in-memory buffer and feed it to the compression algorithm. We selected the `lineitem` table because it is the largest table in TPC-H and efficient (de)compression is critical for its performance. Although compression in Parquet files operates at the granularity of pages, in this microbenchmark we configured the compression algorithm to use a larger block size, as Deflate-based algorithms tend to perform better with larger inputs. Table 1 summarizes the compression and decompression speed of the `lineitem` table (Original Size 4.8GB) when compressed with various compression algorithms.

As expected, Snappy and LZ4 offer faster decompression speed than zstd, while offering the worst compression ratio. The reason for this behavior is that Snappy and LZ4 are optimized for faster compression and decompression by sacrificing compression ratio. Therefore, zstd offers notably better compression ratios for all its compression levels (1,3,9), but the returns are diminishing with higher levels of compression. With higher compression level (e.g., 9) the compression speed drops significantly and the compression ratio increases slightly. The decompression speed remains almost the same at all levels. Intel QPL offers an implementation of Deflate in CPU (written in C++) but also offers offloading the computation to the accelerator (IAA). The fastest compression speed is achieved when using the IAA, reaching **4.59 GB/s** while the fastest decompression is also achieved by the IAA with **18.24 GB/s**.

Since each algorithm has unique characteristics, it is important to evaluate the performance impact of the IAA by doing an apples-to-apples comparison. To ensure a fair and direct comparison, we also contrast it with the software implementation of Intel QPL. Compared to the CPU execution, IAA's performance is almost ~1.8x faster for compression and almost ~5.5x for decompression. The compression ratio between the Deflate CPU and IAA differs because IAA operates on a different compression level (IAA only supports one compression level). However, the performance gains stem from IAA's hardware specialization for compression (simpler instruction set architecture) and dedication to only executing specific tasks.

Since the Parquet file representing the `lineitem` table is relatively large (4.8 GB), we also ran experiments on the remaining TPC-H tables to evaluate how Deflate and Intel IAA perform on smaller tables and their corresponding Parquet files. Figure 4 shows the compression speed and compression ratio of most TPC-H tables (region and nation have 5 and 25 records, respectively, and are omitted). For all TPC-H tables, IAA provides the fastest compression speed with a competitive compression ratio. Similar to the previous experiment, zstd offers a slightly higher compression ratio at the cost of being the slowest compressor among all other compression algorithms. LZ4 is faster than Snappy in terms of compression speed, but provides a slightly worse compression ratio. Nevertheless, both algorithms provide significantly slower compression speeds than IAA and a worse compression ratio. Finally, Deflate CPU offers a slightly higher compression ratio than IAA since the CPU implementation can work with a different compression level. A limitation of the IAA is that, unlike the CPU implementation, the compression level cannot yet be adjusted at the current version of Intel QPL.

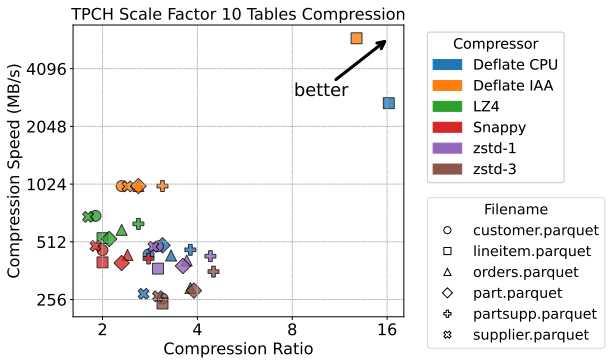

Figure 4: Compression Speed and Ratio of TPC-H Tables in the Parquet Format

Our decompression results indicate similar patterns. Figure 5 captures the decompression speed and compression ratio of the TPC-H tables.

As discussed, zstd provides the highest compression ratio. Yet, in the case of decompression, Deflate CPU is the slowest, followed by zstd. Snappy offers a decompression speed similar to zstd, but its compression ratio is significantly lower. LZ4 and Snappy provide the

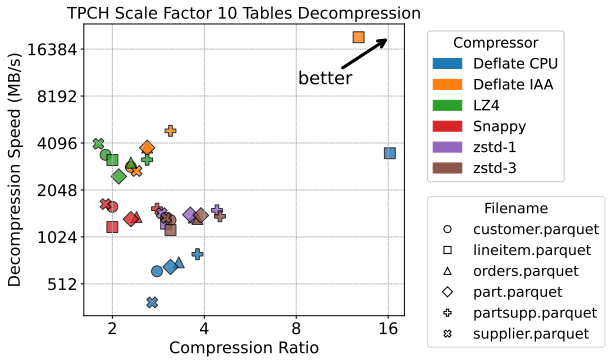

Figure 5: Compression Speed and Ratio of TPC-H Tables in the Parquet Format

lowest compression ratios. On the other hand, LZ4's decompression speed is close to IAA's, making them the two fastest decompressors. Nevertheless, IAA offers a much better compression ratio than LZ4 and strikes a better balance between compression ratio and compression/decompression speed. The outliers in both graphs refer to the `lineitem` table, which is compressed very efficiently by Deflate, both in CPU and IAA.

The purpose of the previous microbenchmark was to examine how well each different compression algorithm handles a large block of data. We found that Deflate works better with larger blocks. However, as described in subsection 2.1, Parquet compresses data at a page's granularity, typically a couple of MBs. The size of each table after being compressed with DuckDB's parquet writer at the granularity of a page is shown in fig. 6 . Similar to the microbenchmark, the compression ratio follows a similar pattern when the tables are compressed in a realistic setting. LZ4 and Snappy consistently offer a lower compression ratio, while the IAA provides a competitive compression ratio close to zstd.

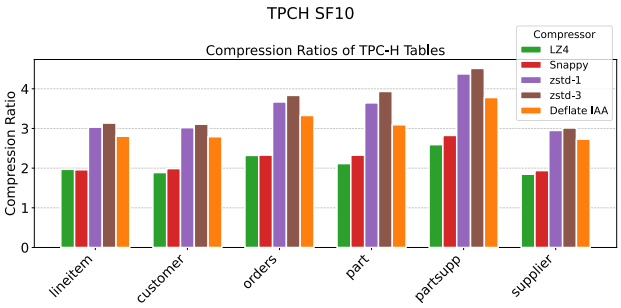

Figure 6: Compression Ratio of each TPC-H Table when compressed with Parquet Writer

Figure 7 shows the end-to-end runtimes for all the TPC-H queries. Similar to the previous experiments, the slowest query runtime is observed when zstd is used for decompression. While zstd offers the highest compression ratio, the queries experience longer runtimes because of zstd's slow decompression speed. Moreover, our experiments indicate that using Snappy as a compression algorithm leads to noticeably slower query runtimes than LZ4 and IAA. Snappy offers a lower compression ratio than LZ4 and IAA. Therefore, even though Snappy is similarly fast to LZ4, its worst compression ratio results in more I/Os, which increase the overall query latency. Finally, IAA is typically faster than LZ4, offering advantages in end-to-end query latency and storage savings.

5.2 Discussion

Intel IAA offers very fast compression and decompression speeds and a competitively high compression ratio. Our early experiments with DuckDB and TPC-H indicate that IAA has a promising future for DBMSs. IAA offers a simple developer experience, which makes it easy to integrate into existing codebases. Moreover, unlike other accelerators or devices (e.g., GPUs), IAA does not require explicit memory management and eliminates the memory movement since it can utilize the CPU's memory subsystem.

However, despite all its advantages, IAA has certain limitations. More specifically, IAA currently only supports a few compression algorithms, making it restrictive for general adoption. For example, the Parquet file format does not officially support the Deflate compression algorithm, and therefore, IAA cannot be used by DBMSs that interact with Parquet files. Therefore, the lack of flexibility to implement various compression algorithms is one of IAA's serious disadvantages.

6 RELATED WORK

Data compression is used in almost all modern DBMSs and it has been widely studied by the database community [1, 6, 12, 16, 27, 29, 31, 32, 36]. Most DBMSs support multiple compression algorithms, each tuned and specialized for different use cases, such as file compression to reduce disk I/Os or in-memory data representations to reduce the memory requirement during query execution and overcome the memory bandwidth bottlenecks [29]. The primary motivation for integrating compression into DBMSs is to mitigate the performance gap between fast CPUs and slower I/O subsystems such as disks and networks, by reducing data volume during storage and transmission. Exchanging relatively inexpensive CPU cycles for reductions in disk I/Os and network traffic is generally advantageous for DBMSs, given the performance gap between compute and I/O subsystems.

Some DBMSs, such as MySQL, use general-purpose byte-agnostic compression algorithms (like LZ4 and ZLIB). These algorithms treat a high compression ratio as a first-class citizen while striving for good (de)compression speeds. With the rise of columnar DBMSs for analytical workloads, "lightweight" compression schemes (encodings) such as dictionary encoding, RLE, and Delta encoding became a common practice [1, 6, 27, 29]. Columnar layouts—where values of the same type are stored contiguously—enable these type-aware encodings to achieve good compression ratios with low computational overhead. Lightweight compression has been studied for query execution in the context of C-Store [1] and is beneficial in many cases, since it can reduce the memory requirements during query execution. With improvements in low-latency and high-bandwidth networks and SSDs, "heavyweight" compression algorithms can

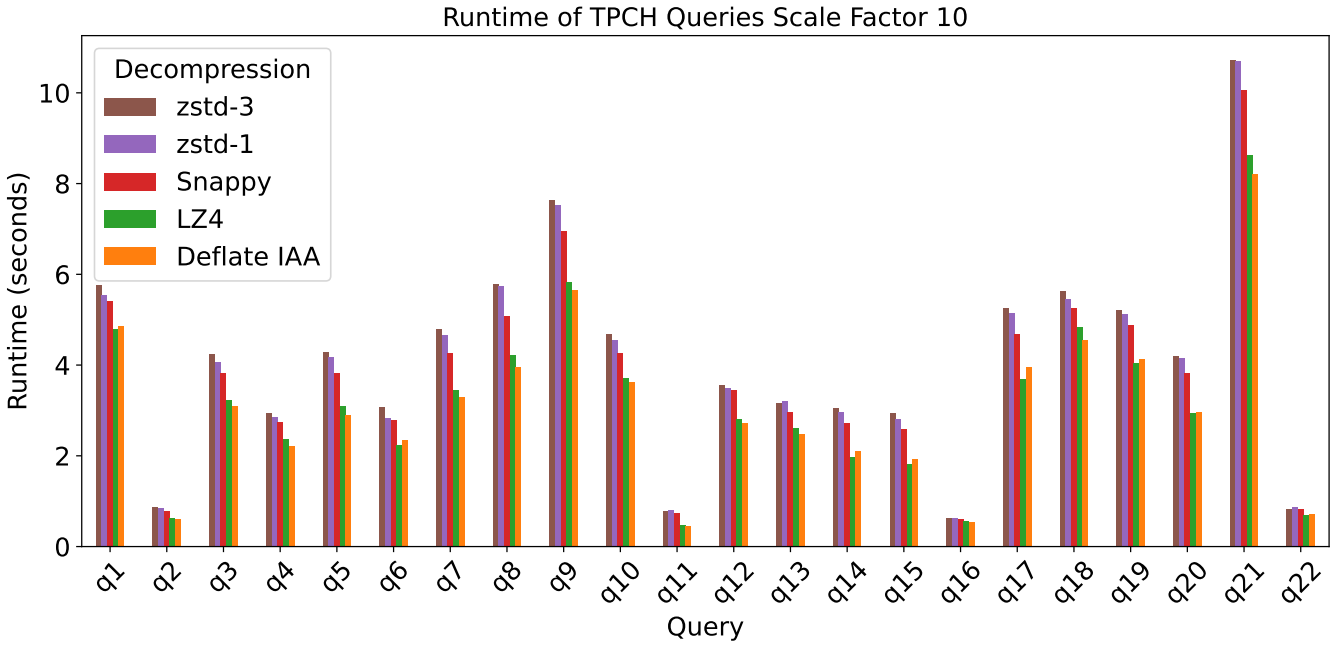

Figure 7: TPC-H scale factor 10 in DuckDB, Single Threaded Execution

become CPU-bound, shifting the performance bottleneck from I/O to compute.

Recently, BtrBlocks [22] identified this bottleneck in the context of data lakes, particularly when operating in high-performance data centers with high-throughput networks. BtrBlocks [22] optimizes for faster compression and decompression speeds while modestly sacrificing the compression ratio, by cascading multiple "lightweight" encodings that are faster to compute and can often be vectorized with SIMD instructions [24, 29]. Fastlanes [2] proposed a new compression layout and accelerated the decoding of lightweight encodings. Moreover, one of these encodings, Bitpacking, can be leveraged to accelerate scans and filters by operating on the bit level and skipping unnecessary computation at runtime [25]. String compression is essential for analytical DBMS since strings take longer to process (compare, hash). FSST [7] is a new "lightweight" compression scheme for strings. It is based on dictionary encoding and builds a symbol table containing common prefixes. Chimp [26] and ALP [3] are schemes that target floating-point numbers. With the recent CPU performance stagnation, another research direction has focused on hardware-accelerated compression [8] via FPGAs.

7 CONCLUSION AND FUTURE WORK

In this work, we present an early examination of Intel IAA, a built-in accelerator recently introduced in the latest generation of Xeon server processors. Our preliminary exploration of (de)compression performance on Parquet files and end-to-end queries in DuckDB suggests that IAA is a promising technology that can deliver notable performance gains, since it offers competitive compression ratios and high compression and decompression speeds. We believe that hardware accelerators like Intel IAA have a promising future and we plan to further investigate opportunities for their integration into DBMSs, especially those that are based on open file formats.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under the grant SHF-2407690. We also thank Peter Zhong and Dimitrios Skarlatos for their insightful conversations and feedback.

REFERENCES

[1] 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

[2] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (May 2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[3] Azim Afroozeh, Leonardo Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proceedings of the ACM on Management of Data* 1 (12 2023), 1–26. https://doi.org/10.1145/3626717

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 169–180.

[5] Apache Software Foundation. 2025. Apache Parquet. https://parquet.apache.org/.

[6] 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

[7] 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

[8] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in SAP HANA. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3277–3291. https://doi.org/10.14778/3554821.3554822

[9] L. Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. https://doi.org/10.17487/RFC1951

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[11] Facebook. 2025. zstd. https://github.com/facebook/zstd/.

[12] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 670–680. https://doi.org/10.14778/1920841.1920927

[13] 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

[14] Google. 2025. LZ4. https://github.com/lz4/lz4/.

[15] Google. 2025. Snappy. https://github.com/google/snappy/.

[16] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. 2021. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS.

[17] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[18] Intel. 2025. accel-config. https://github.com/intel/idxd-config/.

[19] Intel. 2025. Intel QPL. https://github.com/intel/qpl/.

[20] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[21] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTPOLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[22] 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

[23] Juchang Lee, Yong Sik Kwon, Franz Färber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo Yeon Lee, Arthur H. Lee, and Wolfgang Lehner. 2013. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1165–1173. https://doi.org/10.1109/ICDE.2013.6544906

[24] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.* 45, 1 (Jan. 2015), 1–29. https://doi.org/10.1002/spe.2203

[25] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/2463676.2465322

[26] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.* 15, 11 (July 2022), 3058–3070. https://doi.org/10.14778/3551793.3551852

[27] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 283–294. https://doi.org/10.5441/002/EDBT.2014.27

[28] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:209379505

[29] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (Melbourne, VIC, Australia) *(DaMoN'15)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/2771937.2771943

[30] Mark Raasveldt and Hannes Muehleisen. [n.d.]. *DuckDB*. https://github.com/duckdb/duckdb

[31] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast integer compression using SIMD instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (Indianapolis, Indiana) *(DaMoN '10)*. Association for Computing Machinery, New York, NY, USA, 34–40. https://doi.org/10.1145/1869389.1869394

[32] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The implementation and performance of compressed databases. *SIGMOD Rec.* 29, 3 (Sept. 2000), 55–67. https://doi.org/10.1145/362084.362137

[33] 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

[34] 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

[35] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (May 1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[36] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, USA, 59. https://doi.org/10.1109/ICDE.2006.150