On "Hardware Acceleration of Compression and Encryption in SAP HANA"

DT-DB42-M: The Question to or the Better Answer on 42? (SS2023)

Di Xu
University of Bamberg
Bamberg, Bayern, Germany
di.xu@stud.uni-bamberg.de

ABSTRACT

To improve database performance as well as data security, compression and encryption need to be considered in the database management system. In this paper, we introduce the DEFLATE Compression method and the three modes of The Advanced Encryption Standard (AES). Compression and different types of encryption are then composed to suit different levels of security and to determine which combinations maximize performance. Finally, an experimental evaluation using trace data from SAP HANA is performed.

KEYWORDS

compression, encryption, DEFLATE, AES, SAP HANA

1 INTRODUCTION

Security and performance are always two of the most important evaluation criteria when designing a database management system. Data compression and encryption are two techniques that can help to reduce data size and protect data confidentiality[3]. They have different but complementary goals. Compression reduces the space and bandwidth required to store and transmit data, thereby increasing efficiency, speed and cost. Encryption converts data into an unreadable form that can only be decrypted using a key, thus preventing unauthorized access, tampering and leakage. It is clear that with the use of these two technologies, the security and transfer efficiency of data and backup storage performance have been enhanced or optimized.

In this paper, we explore compression and encryption in SAP HANA cloud deployments. SAP HANA is a column-oriented in-memory database specifically built to integrate both analytical and transactional workloads into a single engine. It can be deployed on-premise or as part of SAP HANA Cloud as a SaaS [1]. In Section 2, we introduce the DEFLATE compression algorithm and AES encryption mode, and design a combination of compression and three different encryption modes. In Section 3, we evaluate the performance of the three combinations based on SAP HANA. The results we obtained demonstrate the advantages of these designs: we can efficiently compress and encrypt data as it flows to storage (up to 4 GB/s and 15 GB/s, respectively, and about 4 GB/s when combined).

2 METHOD

In this section, we separately describe the algorithms for compression and encryption, and combine the two designs to form an efficient architecture.

2.1 Compression

Common data compression methods are classified as lossless compression and lossy compression. Lossless compression allows complete restoration of the original data without any loss of information when decompression occurs. Common lossless compression algorithms are: Huffman Coding, LZ77 and LZ78, DEFLATE compression algorithm. Using lossy compression the compressed data is decompressed with some degree of information loss. It is commonly used to compress media data such as audio, images and video to reduce file size while maintaining applicability. Common lossy compression algorithms are JPEG (for image compression), MP3 (for audio compression), MPEG (for video compression). These compression algorithms are selected for use depending on the type of data and application scenario. Lossless compression is suitable for scenarios where complete restoration of the original data is required, while lossy compression is suitable for situations where a certain loss of information is acceptable.

Compressing data on its way to storage is typically done using different methods. Being already in use in SAP HANA, we focus on the DEFLATE method for heavy-weight compression[1].

The DEFLATE compressed data format consists of a series of blocks, corresponding to successive blocks of input data. Each block is compressed using a combination of the LZ77 algorithm and Huffman coding[2]. The basic idea of the LZ77 algorithm is that there are many recurring strings in the data (which can also be byte streams), and the more repetitions, the more compressible space. For these recurring strings, we can use the <length, distance> pair, where distance is the distance to the last occurrence of the string and length is the length of the recurring string. Huffman coding is a prefix coding based on counting the number of occurrences of each character in the entire data to be encoded. A binary tree is dynamically built from the bottom up by sorting characters from smallest to largest according to their frequency. That is, from the bottom up, the leaf nodes of the binary tree are each character, and the fewer occurrences of the character at the bottom of the binary tree, the longer the corresponding length of the code word. Deflate combines the LZ77 algorithm with Huffman coding, first applying the LZ77 compression strategy to compress the original data to get the (<length, distance> or literal) stream, and then applying Huffman coding to encode distance, length, and literal to get the final compressed data stream respectively.

As shown in the Figure 1, the DEFLATE execution is divided into five parts. External memory accesses are handled by dedicated kernels (read data, load Huffman tree, send to DDR4), so resource

allocation can be optimized to accommodate specific read/write kernels and achieve high operation frequency.

The compression of the client payload (i.e., database pages) is split into multiple transactions. Different cores can process different transactions from the same client payload in parallel. The payload is sent to external memory and all kernels receive multiple control parameters, in particular the memory pointer and the size of the input to be processed. Each compute core (LZ77 compression, Huffman encoding) receives a transaction from the input FIFO, processes it, and sends the resulting transaction to the next core via the output FIFO. Execution ends when the memory core (sent to DDR4) finishes writing the last part of the last compressed transaction to main memory.

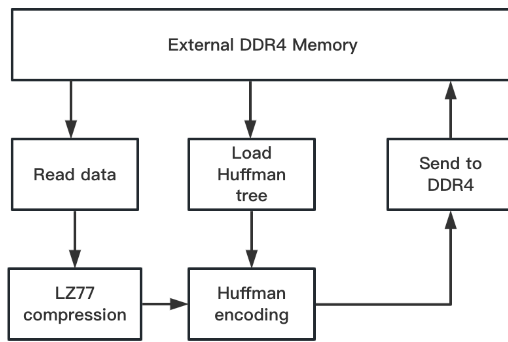

Figure 1: Compression block diagram[1]

2.2 Encryption

The common encryption methods are symmetric encryption and asymmetric encryption. As the name implies, symmetric encryption uses the same key to encrypt and decrypt data, such as AES and DES, while asymmetric encryption uses a public key to encrypt data and a private key to decrypt it, such as RSA.

The Advanced Encryption Standard (AES) is the most widely used packet cipher encryption standard with three possible initial key lengths: 128-bit, 192-bit and 256-bit, and we will focus on the 256-bit implementation. The design of the algorithm is based on a series of substitutions and permutations, called transformation rounds. The number of rounds depends on the key length, 128 bit key – 10 rounds, 192 bit key – 12 rounds, 256 bit key – 14 rounds. The AES encryption algorithm defines numerous transformations that are to be performed on data stored in an array. The first step of the cipher is to put the data into an array, after which the cipher transformations are repeated over multiple encryption rounds(Figure 2). The first transformation in the AES encryption cipher is substitution of data using a substitution table. The second transformation shifts data rows. The third mixes columns. The last transformation is performed on each column using a different part of the encryption key. Longer keys need more rounds to complete.

However, its block cipher modes of operation cause significant performance differences due to the resulting implementations. We compare three AES block cipher modes: Electronic Code Book

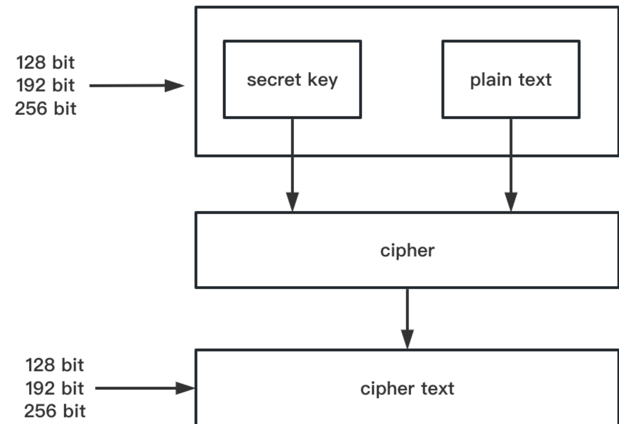

Figure 2: AES workflow

(ECB), Counter (CTR), and Cipher Block Chaining (CBC). ECB mode is the simplest, each block will be encrypted using the same key and the same algorithm. So, if we encrypt the same plaintext, we will get the same ciphertext. So there is a high risk in this mode. And the plaintext and ciphertext blocks are one-to-one correspondence. Since encryption/decryption are independent, we can encrypt/decrypt the data in parallel. If one block of plaintext or ciphertext is corrupted, it will not affect other blocks. In the CTR operation mode, the counter value is used as the input block for the Encrypt. The output block of the Encryptor performs an operation of XOR with the plaintext block. All encryption blocks use the same encryption key. If you can get the counter directly, you can encrypt/decrypt data in parallel. CBC mode is implemented by using an initialization vector (IV). In CBC mode, the IV must be unpredictable (random or pseudorandom) at encryption time. We will use the plaintext block xor with the IV. Then CBC encrypts the result into a ciphertext block. In the next block, we will use the encrypted result to dissociate with the plaintext block until the last block. In this mode, even if we encrypt the same plaintext block, we will get different ciphertext blocks. We can decrypt the data in parallel, but it is not possible when encrypting the data. If a plaintext or ciphertext block is corrupted, it will affect all subsequent blocks.

2.3 Compression and Encryption

We take advantage of the versatility of the OpenCL environment by combining modules written in different languages. The compression module is built in the OpenCL kernel, while the encryption module is built in VHDL but integrated into OpenCL as a library and exposed to the system kernel as a function call. The result is a combination operator that compresses and then encrypts the data when called by the database (Figure 3). When the compression module is used alone, the compression result is sent directly to external memory. When connected to the encryption module, the compressed transactions are forwarded to the AES-256 module via a FIFO buffer.

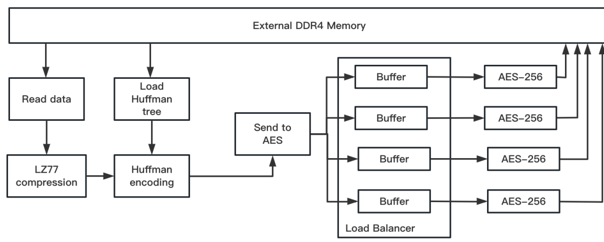

Figure 3: Compression and encryption pipeline block diagram for the three block cipher modes[1]

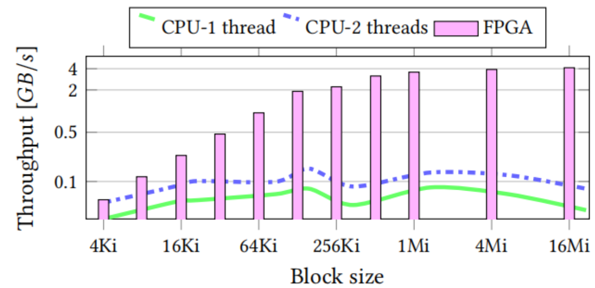

Figure 4: Compression - with 1 and 2 threads on CPU vs. FPGA design[1]

3 EVALUATION

Chiosa, Monica, et al.[1] evaluate the performance of each module in standalone and combined cases. As a benchmark, using typical configurations commonly used in SAP HANA today, and focus on compression and encryption for blocks ranging in size from 4 KiB to 16 MiB.

Software. For the software baseline, use an Intel® Xeon® Gold 6234 Processor 3.3 GHz machine with 8 cores and 16 threads featuring: 512 kB (L1 cache), 8 MB (L2 cache), and 24.75 MB (L3 cache). The level of parallelism set for our compression/decompression and encryption/decryption baselines is consistent with the number of threads SAP HANA allocates for these background tasks, namely 1-2 threads. These threads process the blocks in their entirety.

Hardware. The target platform consists of the Intel Programmable Accelerator Card (PAC) for data centers, Intel FPGA PAC D5005 , connected to the CPU via a PCIe Gen3x16 link. The card features a Stratix 10 SX FPGA, two QSFP+ connectors with up to 100 Gbps support, and 32 GB of on-board DDR4-2400 memory, with a peak transfer rate of 19.2 GB/s. Using OpenCL for Intel FPGA SDK (OpenCL RTE version 19.2.0.57) to implement and instantiate the FPGA compute kernels that are interfacing with the on-board DDR4 memory at 64 B cacheline granularity for both read and write operations. The CPU (host processor) allocates memory for the FPGA computing kernels, and a memory management library handles the address translation between CPU main memory and FPGA external memory.

3.1 Compression

Figure 4 presents the performance comparison of both software baselines and our standalone compression kernel on the FPGA. Compared to the CPU baselines, the FPGA achieves over an order of magnitude speed-up for block sizes larger than 64 KiB. While the overhead costs for small block sizes for the FPGA come from both the memory movement overhead and the Huffman tree; on the CPU, the bottleneck of the cache size cannot be avoided, because it affects the ability of the CPU to efficiently access and retrieve data from memory. Therefore, the throughput gain obtained on the FPGA by increasing the block size is not seen on the CPU, where the fastest implementation saturates at less than 0.2 GB/s.

3.2 Encryption

As a baseline, Chiosa, Monica, et al.[1] build a library on top of Intel AES intrinsic instruction set for the three AES-256 modes (ECB, CTR, CBC). Each block cipher mode receives for encryption/decryption the same block sizes as the ones traced in SAP HANA.

The results show that the three modes differ significantly in terms of software performance. The ECB mode is the simplest and thus performs the best in terms of software performance with a maximum throughput of 4 GB/s for both encryption and decryption. The CTR mode adds complexity with a maximum throughput of 2.5 GB/s. The CBC mode, where the cost of data dependency combined with the cost of heterogeneous operations results in a maximum throughput of 1.2 GB/s.

Figure 5 shows the limitations of the MHz operational clock range of the FPGA. Even if for the FPGA the XOR operation comes at no performance cost, the data dependency translated into the sequential nature[1] of the CBC mode implementation limits the FPGA CBC encryption throughput performance to 0.27 GB/s. At block size granularity, CBC encryption cannot take advantage of the parallelization potential of the FPGA, whereas the CTR and ECB modes benefit from it, reaching a maximum throughput performance of 15 GB/s. By exploiting the spatial parallelism[2] available on the FPGA, CTR and ECB encryption modes exceed by up to seven times their corresponding CPU performance.

3.3 Compression and Encryption

As observed in Figure 4 and Figure 5, the modules have a very different maximum throughput. Compared to the 4 GB/s saturation throughput delivered by the compression module, the CBC mode (0.27 GB/s) would impose back-pressure and limit the overall performance, whereas both CTR and ECB modes (15 GB/s) would turn compression into the bottleneck.

[1]Sequential nature refers to the characteristic of a process or operation that must be executed in a specific order, one step at a time, without parallelization or concurrent execution. In the context of the statement, the sequential nature of the CBC (Cipher Block Chaining) encryption mode means that the encryption process for each block depends on the result of the encryption of the previous block. This dependency creates a sequential chain of operations, where each block must be encrypted in order, one after the other.

[2]Spatial parallelism refers to the simultaneous execution of tasks by several processing units.

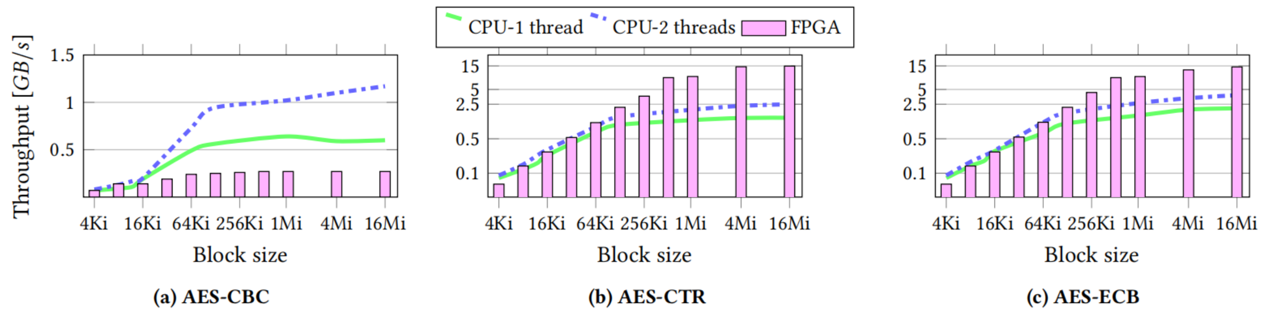

Figure 5: AES Encryption - with 1 and 2 threads on CPU vs. FPGA design[1]

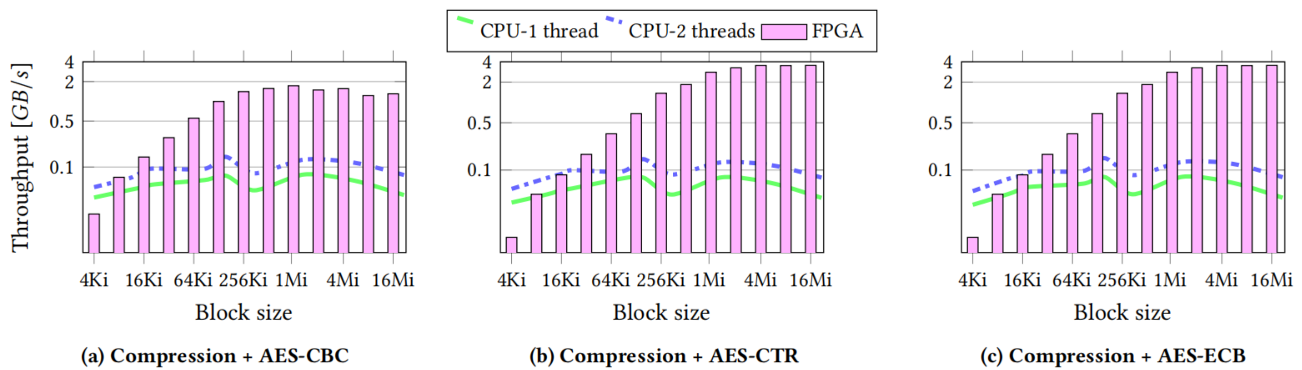

Figure 6: Full pipeline - with 1 and 2 threads on CPU vs. FPGA design[1]

In Figure 6 we analyze its throughput performance. A maximum throughput of 1.72 GB/s was achieved with CBC mode encryption, while for CTR or ECB mode encryption, a level of 4 GB/s was achieved comparable to the throughput imposed by the compression module alone.

4 CONCLUSION

In this paper, we present algorithms for encryption and compression and evaluate them in their individual and combined forms. The results show that the maximum throughput of both DEFLATE compression + CTR/ECB converges to 4 GB/s greater than the 1.92 GB/s of the DEFLATE compression + CBC combination when the data size is large enough. But the CBC model adds an extra level of complexity to the encrypted data, which makes the data more secure. Exactly how encryption and compression be combined could be considered in more ways than just throughput. Including the order of encryption and compression will also have an impact on the encryption and compression process, which may be a perspective that can be studied in the future.

REFERENCES

[1] Chiosa, Monica, et al. "Hardware acceleration of compression and encryption in SAP HANA." 48th International Conference on Very Large Databases (VLDB 2022). 2022.

[2] Oswal S, Singh A, Kumari K. Deflate compression algorithm[J]. International Journal of Engineering Research and General Science, 2016, 4(1): 430-436.

[3] How Do You Balance Data Security and Performance When Compressing and Encrypting Data? (2023, June 22). Linkedin. https://www.linkedin.com/advice/1/how-do-you-balance-data-security-performance