Improving Relational Database Upon the Arrival of Storage Hardware with Built-in Transparent Compression

1st Yifan Qiao
Rensselaer Polytechnic Institute

2nd Xubin Chen
Rensselaer Polytechnic Institute

3rd Jingpeng Hao
Rensselaer Polytechnic Institute

4th Jiangpeng Li
ScaleFlux Inc.

5th Qi Wu
ScaleFlux Inc.

6th Jingqiang Wang
ScaleFlux Inc.

7th Yang Liu
ScaleFlux Inc.

8th Tong Zhang
Rensselaer Polytechnic Institute

***Abstract*—This paper presents an approach to enable relational database take full advantage of modern storage hardware with built-in transparent compression. Advanced storage appliances (e.g., all-flash array) and some latest SSDs (solid-state drives) can perform hardware-based data compression, transparently from OS and applications. Moreover, the growing deployment of hardware-based compression capability in Cloud storage infrastructure leads to the imminent arrival of cloud-based storage hardware with built-in transparent compression. To make relational database better leverage modern storage hardware, we propose to deploy a dual in-memory vs. on-storage page format: While pages in database cache memory retain the conventional row-based format, each page on storage devices has a column-based format so that it can be better compressed by storage hardware. We present design techniques that can further improve the on-storage page data compressibility through additional light-weight column data transformation. We the impact of compression algorithms on the selection of column data transformation techniques. We integrated the design techniques into MySQL/InnoDB by adding only about 600 lines of code, and ran Sysbench OLTP workloads on a commercial SSD with built-in transparent compression. The results show that the proposed solution can bring up to 45% additional reduction on the storage cost at only a few percentage of performance degradation.**

I. Introduction

This paper studies how relational database can take full advantage of modern storage hardware with built-in transparent compression capability. Modern all-flash array products (e.g., Dell EMC PowerMAX [9], HPE Nimble Storage [13], and Pure Storage FlashBlade [21]) have built-in hardware-based transparent compression. SSDs with built-in transparent compression are emerging on the commercial market (e.g., computational storage drive from ScaleFlux [23] and Nytro SSD from Seagate [12]). Cloud vendors have started to integrate hardware-based compression capability into cloud infrastructure (e.g., Microsoft Corsia [7] and AWS Graviton2 [3]), leading to imminent arrival of cloud-based storage hardware with built-in transparent compression. Although most relational database (e.g., MySQL and Oracle) incorporate record/page-level compression capability, it is not uncommon that users disable such feature in order to avoid performance degradation. Hence, when running on such modern storage hardware, relational database can benefit from storage cost reduction without sacrificing performance and CPU usage.

This work is interested in how one could make relational database gain more storage cost reduction from such storage hardware. The key is to increase the database compressibility at minimal impact on the database performance. In order to well serve OLTP (on-line transactional processing) workloads, most relational databases use the convenient row-based page format, with the typical page size of 8KB or 16KB. Meanwhile, it is well known that, by clustering the same-type data together, column-based storage format [1] enables better data compressibility than row-based format. This motivates us to study the potential of deploying dual page format in relational database: When a page resides in database cache memory, it keeps the conventional row-based format, and all the on-storage pages have a column-based format. When database fetches a page from storage or flushes a page to storage, it accordingly carries out on-the-fly row-column format conversion. Such dual in-memory vs. on-storage page format can largely enhance the storage cost saving and meanwhile keep the database query processing engines completely intact.

Beyond the row-column conversion, we apply certain column data transformation to further boost the data compressibility. To minimize the impact on database performance, we propose several light-weight data transformation schemes that can leverage SIMD (single instruction, multiple data) CPU instructions. We note that column data transformation has a fundamentally different purpose from column data encoding (e.g., dictionary and run-length encoding) in column-store: The former aims at improving the column data compressibility so that the storage hardware can achieve better compression ratio, while the latter aims at directly compressing the column data. Storage hardware with built-in transparent compression always performs general-purpose LZ-family compression that either does LZ-search only (e.g., lz4 [17]) or concatenates LZ-search with entropy coding (e.g., ZSTD [30] and zlib [29]). We

978-1-7281-7744-1/21/$31.00

show that different type of LZ-family compression algorithms may favor different data transformation schemes. For the purpose of evaluation, we modified the InnoDB storage engine in MySQL 8.0 to support the proposed dual page format, by adding only about 600 lines of code. We ran Sysbench OLTP workloads on a commercial 3.2TB SSD with built-in transparent compression [23]. The results show that our proposed techniques can improve the storage cost savings by up to 45% and meanwhile only incur a few percentage MySQL TPS (transactions per second) performance loss. In summary, this paper makes the following main contributions:

1) We propose a dual in-memory vs. on-storage page format design framework that enables relational database take full advantage of the growing family of storage hardware with built-in transparent compression;
2) We develop light-weight column data transformation that can largely improve the on-storage page data compressibility with very small impact on database performance;
3) We analyze the impact of compression algorithms being implemented in storage hardware on the selection of column data transformation;
4) Using MySQL as a test vehicle, we demonstrated the effectiveness of this proposed approach on a commercial SSD with built-in transparent compression.

II. Background

A. Compression in Relational Database

In order to better serve OLTP workloads, mainstream relational database by default uses B-tree index structure in which each leaf page stores data tuples in the row-based format. Relational database typically provides users with different compression options (e.g., row/prefix/dictionary compression in SQL Server, and table/page compression in MySQL). However, compression inevitably incurs CPU overhead and longer latency, leading to database performance degradation. For the demonstration, on a 2U server with 32-core Xeon CPU, we ran MySQL 8.0 with Sysbench OLTP benchmarks (32 client threads in all the experiments). Fig. 1 shows the TPS performance under three different configurations: (1) the default *no compression*, and (2) lz4-based page compression, and (3) zlib-based page compression. Compared with zlib compression, lz4 compression consumes less CPU resource at the loss of data compression ratio. The results show that page compression can significantly degrade MySQL performance, especially under relatively write-intensive workloads. Therefore, it is not uncommon that users turn off the database compression in spite of good user data compressibility.

B. Storage-level Transparent Compression

By realizing data compression at the storage level (e.g., filesystem or block device), we can allow all the upper-level user applications to benefit from storage cost reduction without explicitly handling data compression. Industry has integrated transparent compression into many storage-level software. For example, ZFS [5] and Btrfs [22] support filesystem-level transparent compression, latest Red Hat Enterprise Linux contains a VDO (Virtual Data Optimizer) module that realizes block-level transparent compression. However, all the software solutions consume CPU resource to implement and management data compression, which makes the system still subject to the performance vs. storage cost trade-off.

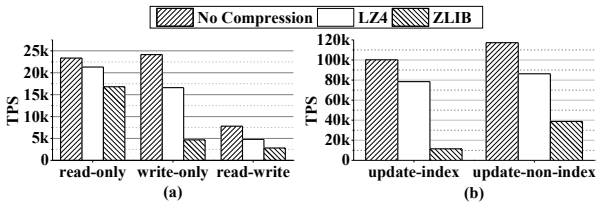

Fig. 1. MySQL TPS under Sysbench OLTP benchmarks under three different configurations.

If we can migrate data compression into the storage hardware, systems will be free from the performance vs. storage cost trade-off. Modern all-flash arrays integrate hardware-based block-level transparent compression. Another option is to directly push compression into each individual storage device (e.g., SSD or HDD). Fig. 2 illustrates the structure of an SSD with built-in transparent compression. Data (de-)compression are carried out directly on the IO path by the hardware engine inside the SSD controller, and the FTL (flash translation layer) inside the SSD controller manages the mapping/indexing of all the variable-length compressed data blocks. Such SSDs [23], [12] are now emerging on the commercial market. Moreover, by deploying hardware-based compression capability [7], [3], Cloud data storage infrastructure becomes ready to support transparent compression. Regardless of the specific implementation, storage hardware should always compress each 4KB sector individually in order to avoid random IO performance degradation.

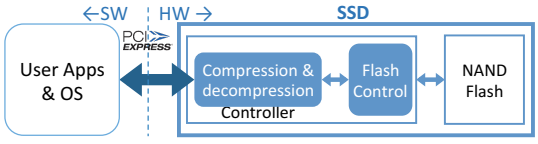

Fig. 2. Illustration of an SSD with built-in transparent compression.

III. Proposed Design Solutions

When running on storage hardware with built-in transparent compression, relational database can seamlessly benefit from storage cost reduction without any source code changes. This work studies the potential of further reducing the storage cost by slightly modifying relational database. Motivated by the success of column-store, this work centers around the simple idea of deploying a dual in-memory vs. on-storage page format as illustrated in Fig. 3: When a page (with the typical size of 8KB or 16KB) resides in the database cache memory (e.g. buffer pool in MySQL), it retains the conventional row-based

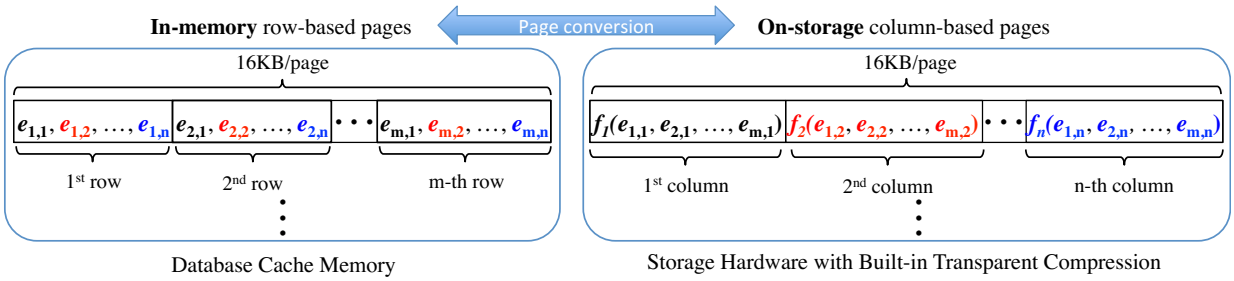

Fig. 3. Illustration of the proposed dual in-memory vs. on-storage page format.

format. When residing on the storage devices, each page has a column-based format. Moreover, we may apply additional data transformation functions (e.g., f_i for the i-th column as illustrated in Fig. 3) to further improve the data compressibility of on-storage pages. Different from data encoding in column-store, the function f_i does not try to reduce the column data size at all. Instead, its goal is to CPU-efficiently manipulate data content to improve its compressibility.

To support dual page format, relational database must carry out on-the-fly page content conversion. When database flushes a page from its cache memory to storage, the page content conversion consists of two steps: (1) *Row-to-column conversion*: Let n denote the number of table fields, m denote the number of rows in one page, and $e_{i,j}$ denote the j-th element of the i-th row in one page. We convert the page from the row-based layout $[R_1, R_2, \cdots, R_m]$, where each row $R_i = [e_{i,1}, e_{i,2}, \cdots, e_{i,n}]$, to a column-based layout $[C_1, C_2, \cdots, C_n]$, where each column $C_j = [e_{1,j}, e_{2,j}, \cdots, e_{m,j}]$. (2) *Column data transformation*: We further apply a transformation function f_j to the j-th column so that the transformed column $f_j(e_{1,j}, e_{2,j}, \cdots, e_{m,j})$ has a higher compressibility than the original column $[e_{1,j}, e_{2,j}, \cdots, e_{m,j}]$. When database fetches one page from storage into its cache memory, it performs the reverse conversion, i.e., first apply the reverse transformation $f_j^{(-1)}$ on each column, and then carry out column-to-row conversion to reconstruct the original row-based page. In spite of the simple idea underlying the dual page format design approach, its practical implementation faces the following two conflicting objectives:

1) *Reduce database performance degradation*: By carrying out data conversion/transformation on the page IO path, it consumes extra CPU cycles and hence causes longer page IO latency, which may degrade the database performance. Aiming to largely retain the database performance, we should reduce the CPU overhead as much as possible.
2) *Improve page data compressibility*: Storage cost reduction solely depends on how well the page content conversion/transformation can improve data compressibility. The more complicated each column data transformation function f_i is, the more it can improve the column data compressibility, but consumes more CPU cycles.

To assist relational database better explore trade-offs between the above two conflicting objectives, we further present techniques that improve the on-storage page compressibility with different compressibility improvement vs. CPU overhead trade-off. Most storage hardware with built-in transparent compression compress each 4KB sector using LZ-family algorithms, under which *data compressibility* is determined by two types of data redundancy: (i) *repeated byte-string redundancy* that is measured by the length and occurrence frequency of repeated byte-strings, and (ii) *entropy redundancy* that is measured by the distribution of the entropy of different byte symbols. Accordingly, there are two categories of LZ-family algorithms: (1) LZ-only algorithms (e.g., lz4 and Snappy) that only exploit the repeated byte-string redundancy: They only carry out *LZ-search* that searches for repeated byte-strings and replaces them with pointers. (2) LZ+EC algorithms (e.g., zlib and ZSTD) that exploit both types of redundancy: They concatenate LZ-search with additional entropy coding (EC), e.g., Huffman [14] or arithmetic coding [26]. These two categories of LZ-family algorithms represent different trade-offs between implementation cost and compression ratio. Subject to different cost constraint, storage hardware may choose to implement different LZ-family algorithms.

As a result, which compression algorithm the storage hardware implements can influence how we should apply specific techniques for improving page data compressibility. By clustering each column data together, row-to-column conversion may improve the repeated byte-string redundancy. Since storage hardware compress each 4KB sector individually and one database page is typically 8KB or 16KB, row-to-column conversion may also improve the entropy redundancy if different columns have largely different byte content statistics. The second step (i.e., column data transformation) should be realized differently when we use different LZ-family compression algorithms. For the purpose of illustration, let us consider the following two examples that use lz4 and zlib to represent LZ-only and LZ+EC algorithms:

1) Let d denote a 1-byte constant (e.g., 0xFF or 0x00). Given a randomly generated data block, we set a certain percentage of bytes at random locations to the constant value d. Fig. 4 shows the measured data compression ratio under LZ-only and LZ+EC algorithms when varying the constant-byte percentage from 10% to 60%. Let l_{orig} and l_{comp} denote the size of the original and compressed blocks, we define *compression ratio* as $\alpha = l_{orig}/l_{comp} \geq 1$. As the constant-byte percentage increases, the byte

symbol with the constant value will have a larger entropy than the other byte symbols. This can benefit LZ+EC compression more than LZ-only compression, as shown in Fig. 4, because of the explicit use of entropy coding in LZ+EC compression.

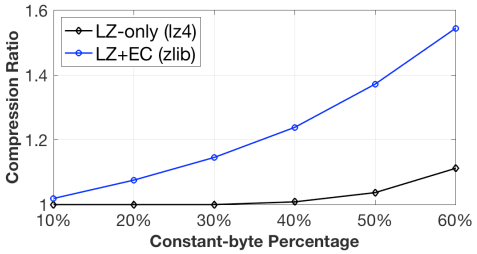

Fig. 4. Measured compression ratio under different constant-byte percentage in randomly generated data blocks.

2) We fill each data block with $[\mathbf{D}_s, \mathbf{R}_s]$, where $\mathbf{D}_s$ represents an s-byte vector $[d, d, \cdots, d]$ (d represents a 1-byte constant) and $\mathbf{R}_s$ represents an s-byte random vector. Hence, within the data block, 50% have the same constant value d, and the other 50% have random values. Almost all the repeated byte-strings are the s-byte vector $\mathbf{D}_s$. Fig. 5 shows the measured compression ratio when we vary the repeated vector length s from 2 to 24. By increasing the value of s, we increase the repeated byte-string redundancy. The results show that higher repeated byte-string redundancy benefit LZ-only algorithm more than LZ+EC algorithms.

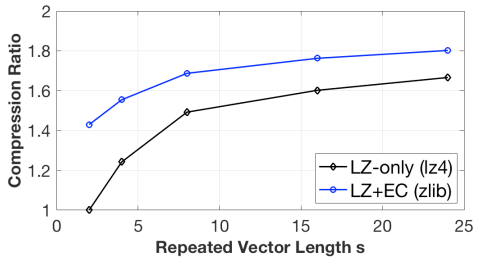

Fig. 5. Measured compression ratio under different repeated vector length s in data blocks with 50% random content.

The above results suggest that, when storage hardware implements different type of LZ-family algorithms, the column data transformation may favor different strategies, which will be further discussed in the remainder of this subsection.

1) LZ-only Compression: When storage hardware implements an LZ-only algorithm, the column data transformation should mainly focus on improving the repeated byte-string redundancy. In particular, we should transform the column data in order to create longer repeated byte-strings. The obvious option is to design the column data transformation following a *shuffling*-centric approach. As illustrated in Fig. 6, its basic idea can be described as follows: Given a byte sequence $[d, d, r_1, r_2, d, d, r_3, r_4, d, d, r_5, r_6, \cdots]$, where d represents a 1-byte constant, each r_i is a random byte, and the 2-byte string

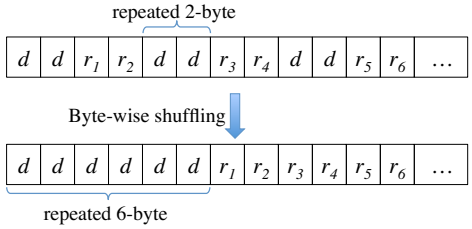

Fig. 6. Illustration of applying byte-wise shuffling to improve the repeated byte-string redundancy.

d's together, we can increase the length of the repeated byte-string, e.g., from repeated 2-byte string to repeated 6-byte string as illustrated in Fig. 6. This will lead to a better LZ-only compression ratio. For its practical implementation, we should use the AVX2 *shuffle* instructions (e.g., *_mm_shuffle_epi8* that can realize user-defined byte-wise shuffling on 128-bit data). Accordingly, we have the following two data transformation strategies:

- *Register-level data transposition*: If the data column has a strong byte-position content locality (i.e., consecutive data elements more likely have the same value at certain byte positions), we can utilize the AVX2 *shuffle* instructions to create longer repeated byte-strings. Suppose each data element contains b bytes, and one register can hold up to k data elements. Let d_i denote the i-th data element, and $d_{i,k}$ denote its k-th byte. Using the AVX2 shuffle instructions, we *transpose* each group of k consecutive data elements $[d_1, d_2, \cdots, d_k]$ into $[t_1, t_2, \cdots, t_b]$, where each t_i consists of k same-position bytes $[d_{1,i}, d_{2,i}, \cdots, d_{k,i}]$. By clustering all the k same-position bytes together within each group of k data elements, such register-level data transposition could create longer repeated byte-strings in the case of strong byte-position content locality.
- *Column-level data transposition*: The above register-level data transposition creates longer repeated byte-string within each register (up to 256-bit). If the data column has a very strong byte-position content locality, one could expand the data transposition to the column level in order to further increase the compressibility. Given a column that consists of total m data elements $[d_1, d_2, \cdots, d_m]$ (where m is larger or much larger than k), we can transpose the entire data column into $[t_1, t_2, \cdots, t_b]$, where each vector t_i consists of m same-position bytes $[d_{1,i}, d_{2,i}, \cdots, d_{m,i}]$. Compared with the register-level transposition, it could potentially create much longer repeated byte-strings.

2) LZ+EC Compression: For storage hardware that uses LZ+EC compression algorithm such as zlib, we should cohesively consider both entropy redundancy and repeated byte-string redundancy. To improve the repeated byte-string redundancy, we can apply the same strategies discussed above (i.e., register-level or column-level data transposition). As shown earlier, compared with increasing the repeated byte-string re-

dundancy, increasing the entropy redundancy tends to be more beneficial to LZ+EC compression. To increase the entropy redundancy, the key is to increase the occurrence frequency of one or few byte values. For data columns with strong byte-position content locality, we can apply byte-wise XOR to generate more bytes with the content of zero, leading to a higher entropy redundancy. Meanwhile, modern CPUs can easily support highly parallel XOR operations.

To improve the data compressibility, we may integrate the byte-wise XOR with register-level data transposition (or the column-level data transposition). Suppose we can load k data elements $[d_1, d_2, \cdots, d_k]$ from one data column into a register. We first carry out byte-wise XOR across all the k data elements to generate a new vector $[d'_1, d'_2, \cdots, d'_k]$, where $d'_1 = d_1$ and $d'_{i+1} = d_i \oplus d_{i+1}$. Given strong byte-position content locality, the new vector may contain a large number of zeros, leading to a higher entropy redundancy. Then we apply the register-level data transposition to further improve the data compressibility by increasing the repeated byte-string redundancy.

IV. Evaluation

Using MySQL as a test vehicle, we performed experiments to evaluate the proposed design solutions. We use a 2U server with 32-core 3.3GHz Xeon CPU and 64GB DRAM. We ran all the experiments on one 3.2TB SSD with built-in transparent compression, which was recently launched to the market by ScaleFlux [23]. This SSD carries out hardware-based per-4KB zlib compression on the IO path. Operating with PCIe Gen3×4 interface, this SSD can achieve 3.2GB/s sequential throughput, and 650K (520K) random 4KB read (write) IOPS (IO per second) over 100% LBA span, which are similar or higher than leading-edge commodity NVMe SSDs.

A. Low-level Evaluation

We first carried out experiments to evaluate the effectiveness of each individual approach on improving the page data compressibility and the corresponding latency overhead. Based on MySQL 8.0, we implemented a MySQL/InnoDB page generator that can generate 16KB pages for any arbitrary table schema and data content characteristics. Each 16KB page contains all the InnoDB page metadata including page header, trailer, and row headers. For the purpose of comparison, we considered the following six different scenarios:(1) *Conventional*: We use the conventional practice as the baseline, where all the 16KB pages keep the same row-based format; (2) *R2C-only*: We only implement the row-column conversion for each 16KB page; (3) *R2C-RLT*: After converting each page from row-based to column-based, we further carry out register-level data transposition; (4) *R2C-CLT*: After converting each page from row-based to column-based, we further carry out column-level data transposition; (5) *R2C-XOR*: After converting each page from row-based to column-based, we further carry out register-level byte-wise XOR (i.e., byte-wise XOR across adjacent column data elements that can fit into one 128-bit register); (6) *R2C-XOR-RLT*: After converting each page from row-based to column-based, we further carry out register-level byte-wise XOR and register-level data transposition.

1) Page Conversion Latency: We first studied the page content conversion latency overhead. The row-column conversion mainly involves memory copy. Hence, the latency of row-column conversion strongly depends on the size of each data item. Regarding column data transformation, its latency depends on the specific transformation processing. Fig 7 shows the measured page conversion latency when each row contains 32 2-byte small integers, 32 4-byte integers, or 32 8-byte big integers. Following the convention in MySQL/InnoDB, we fill each 16KB page up to $\frac{15}{16}$-full and leave the rest $\frac{1}{16}$ as all-zero. We measured the latency of both *forward* conversion from in-memory format to on-storage format and *reverse* conversion from on-storage format to in-memory format. The results show that the *forward* and *reverse* page conversions always have almost the same latency. This is because the row-column conversion and all the column data transformations have very symmetric operations. This is in sharp contrast to all the classical compression/encoding techniques that tend to have very asymmetric operations.

As shown in Fig 7, the latency of row-column conversion (i.e., R2C-only) strongly relies on the data item size, e.g., the latency reduces from 16.7μs in the case of small integers to 6.6μs in the case of big integers. All the XOR and data transposition operations utilize the SIMD instructions. The results show that the latency of column data transformation is largely independent from the data item size. Byte-wise XOR incurs less latency than data transposition. On top of the row-column conversion, the byte-wise XOR incurs about 3μs additional latency, while the register-level data transposition incurs about 5μs additional latency. Compared with register-level data transposition (R2C-RTL), column-level data transposition (R2C-CTL) involves extra memory copy operations in order to spread each transposed 128-bit content across the entire column. The extra memory copy operations incur relatively longer latency. As a result, on top of the row-column conversion, column-level transposition incurs about 15μs additional latency. By concatenating byte-wise XOR and register-level data transposition, R2C-XOR-RTL incurs about 8μs additional latency on top of the row-column conversion.

Let τ_{io} denote the latency of database page IO in current practice, and τ_{conv} denote the extra latency induced by the page content conversion in support of the dual page format. Hence, the impact of dual page format on the overall database performance depends on the τ_{conv} vs. τ_{io} comparison. Under very low SSD IO queue depth (e.g., 1 or 2), the value of τ_{io} is roughly equal to the NAND flash memory access latency that is around 100μs. As the SSD IO queue depth increases under heavier IO stress, the value of τ_{io} can be multiple times longer than the NAND flash memory access latency. Based on the above results, we can draw the conclusion that τ_{conv} tends to be significantly less than τ_{io}.

2) Page Data Compressibility: We further studied the page data compressibility, where lz4 and zlib are used to represent LZ-only and LZ+EC compression. When generating each

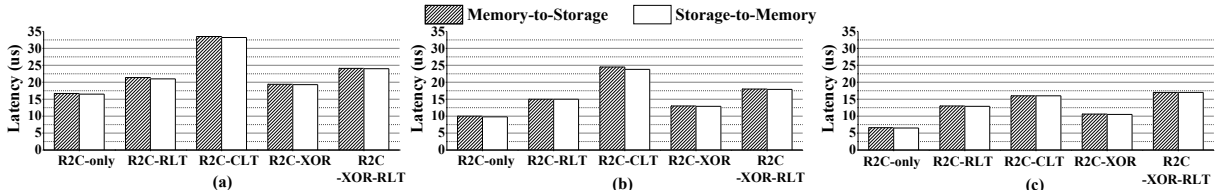

Fig. 7. Measured 16KB InnoDB page conversion latency when each row contains 32 (a) 2-byte small integers, (b) 4-byte integers, and (c) 8-byte big integers.

16KB InnoDB page, we fill the page up to $\frac{15}{16}$-full and leave the rest $\frac{1}{16}$ as all-zero. To better understand the impact of different data types, we studied the data compressibility of the following two categories of pages, each of which contains one data type.

A. *Integer pages*: Each row contains 32 4-byte unsigned integers. For the i-th column, we generate a random integer r_i, based on which we randomly generate all the unsigned integers independently in the i-th column within the $\pm 10\%$ of r_i (i.e., within the range of $[0.9 \cdot r_i, 1.1 \cdot r_i]$). To cover different byte-position content locality, we considered two cases: (i) the magnitude of each r_i falls between 2^{16} and 2^{24}, referred to as *large-integer* page, and (ii) the magnitude of each r_i falls between 2^8 and 2^{16}, referred to as *small-integer* page. Fig 8(a) and (b) show the measured compression ratio of large-integer and

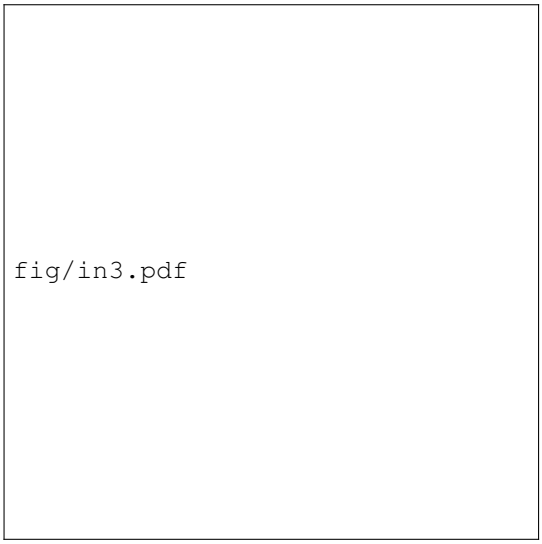

Fig. 8. Measured compression ratio of (a) large-integer pages, and (b) small-integer pages.

small-integer pages, where *Con.* denotes the baseline conventional practice without using the dual page format. The results reveal the following observations: (1) Row-column conversion itself cannot improve the data compressibility at all for integer pages. Due to the randomness in the data generation, clustering the same-column integers together does not bring additional data content redundancy. (2) The comparison between register-level data transposition (R2C-RLT) and byte-wise XOR (R2C-XOR) varies when different compression algorithm is used. R2C-RLT achieves better compression ratio under lz4, while R2C-XOR achieves better compression ratio under zlib. This well matches to the discussion on lz4 vs. zlib comparison in Section III. (3) Column-level data transposition (R2C-CLT) can always achieve the best compression ratio, followed by the concatenation of register-level data transposition and byte-wise XOR (R2C-XOR-RLT). This is because column-level data transposition can most improve the repeated byte-string redundancy. (4) Small-integer pages can benefit more from the column data transformation, because of the stronger byte-position content locality in small-integer columns.

B. *String pages*: Since string data typically do not have strong byte-position content locality, we only applied row-column conversion without any further transformation. We studied the compression ratio of three different types of string pages: (1) *fixed-length string page*: Each row contains 32 fixed-length 16-byte strings. (2) *variable-length string page*: Each row contains 32 strings, where the length of each string is chosen randomly between 8-byte and 24-byte. (3) *mixed-length string page*: Each row contains 16 fixed-length 16-byte strings and 16 variable-length strings, where the length of each variable-length string is chosen randomly between 8-byte and 24-byte. We use the Silesia corpus [10] to generate each string column as follows: First we randomly pick one file from four corpus files (i.e., *dickens*, *samba*, *osdb*, and *webster*), and then randomly choose one contiguous segment from the file to fill the string column. Fig. 9 shows the measured compression ratio of the three different types of string pages. Since string data tend to have relatively high entropy redundancy, zlib achieves significantly higher compression ratio than lz4, especially in the context of column-based string pages. Compared with the conventional row-based page format, column-based page format can improve the compression ratio by up to 35% (zlib) and 27% (lz4). Results also show that variable-length string pages have worse compression ratio than fixed-length string pages, which is mainly due to the impact of row headers.

B. System-level Evaluation

To evaluate the impact on relational database TPS/latency performance, we modified the InnoDB IO process (by adding only about 600 lines of code) so that it can handle the page conversion on the page read and write path. All the modifications are confined in the IO process, being invisible to other modules in MySQL. We ran three Sysbench OLTP workloads (including *write-only*, *read-only* and *read-write*) on a 160GB database. Instead of using the default Sysbench table

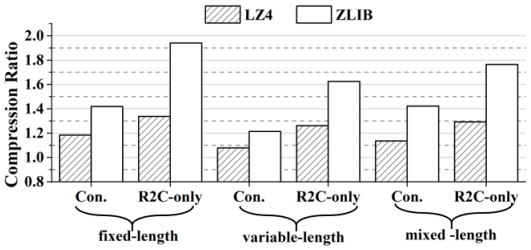

Fig. 9. Measured compression ratio of three different types of string pages.

schema with only 4 fields, we used a table schema consisting of 32 unsigned integer fields and one 32-byte string field. All the integer columns are generated using the same method for generating the small-integer pages in Section IV-A2, and the content of the string column is loaded from the file *webster* in the Silesia corpus. In all the experiments, InnoDB buffer pool is 10GB and InnoDB operates with direct-io, and we kept all the other configurations as their default settings.

For the purpose of comparison, we evaluated the MySQL TPS and average latency performance for all the six scenarios considered in Section IV-A, including *conventional* as the baseline, *R2C-only* that implements row-column conversion only, *R2C-RLT* that implements additional register-level transformation, *R2C-CLT* that implements additional column-level transformation, *R2C-XOR* that implements additional XOR, and *R2C-XOR-RLT* that combines XOR and register-level transformation. For each Sysbench workload, we repeated the experiment with three different number of client threads (i.e., 8, 16, and 32) in order to cover a wide range of workload stress. Fig. 10 shows the measured MySQL TPS and average latency under all the different scenarios and Sysbench OLTP benchmarks. Each data point was obtained by running MySQL about 1 hour. Results show that the proposed approach only slightly degrades the MySQL TPS/latency performance, and the performance impact is almost independent from the client thread number. Under the *write-only* workload, compared with the baseline, *R2C-only*, *R2C-RLT* and *R2C-XOR-RLT* degrade the TPS by 5.2%, 8.1%, and 9.9%, respectively, and increase the average latency by 5.5%, 7.6%, and 10.8%, respectively. Under the *read-only* workload, *R2C-only*, *R2C-RLT* and *R2C-XOR-RLT* degrade the TPS by 1.5%, 3.4%, and 3.7%, respectively, and increase the average latency by 1.3%, 3.4%, and 3.9%, respectively. The results suggest that the proposed design affects read-intensive workloads more slightly than write-intensive workloads. This can be explained as follows: Since the database size is much larger than the InnoDB buffer pool size, MySQL serves each write query by issuing both page fetching and page flushing requests. As a result, the InnoDB IO process will carry out more page conversion operations on the per query basis. In comparison, MySQL serves each read query by only issuing page fetching requests, hence the InnoDB IO process will carry out relatively less amount of page conversion operations. As demonstrated above in Section IV-A1, storage-to-memory and memory-to-storage page conversions have almost the same latency. Therefore, the proposed approach causes relatively larger performance degradation to write-intensive workloads.

The impact of different column data transformation strategies on TPS and latency well matches to their operational latency as shown above in Section IV-A1. With the longest operational latency, *R2C-CLT* corresponds to the largest TPS performance impact, i.e., TPS degradation of 11.2%, 3.9%, and 8.9% under *write-only*, *read-only*, and *read-write* workloads, respectively. In comparison, *R2C-XOR* corresponds to the smallest TPS performance impact, i.e., TPS degradation of 7.0%, 3.3%, and 5.8% under *write-only*, *read-only*, and *read-write* workloads, respectively. Moreover, the results show that, under the *write-only* workload, MySQL TPS and latency are roughly proportional to the number of client threads. In contrast, under the *read-only* workload, the TPS with 16 client threads is closer to that with 32 client clients, while the latency with 16 client threads is almost the same as that with 8 client threads. This is mainly because, under read-intensive workloads, 16 client threads can generate enough read IO requests to almost fully exploit the parallelism among all the NAND flash memory dies inside the SSD. As a result, further increasing the number of client threads from 16 to 32 cannot proportionally improve the TPS, and meanwhile will cause a deeper IO queue depth, leading to a longer read latency. Moreover, we also collected the CPU usage during all the experiments, and the results show negligible CPU usage difference among all the different scenarios.

As mentioned above, we ran all the experiments on a commercial 3.2TB SSD with built-in transparent compression. This SSD internally implements per-4KB zlib compression, and can report the average compression ratio of all the data being stored on the drive. Fig. 11 shows the compression ratio reported by the drive when generating the same 160GB database under all the six different scenarios. For the purpose of comparison, Fig. 11 also includes the compression ratio achieved by MySQL's own lz4/zlib-based page compression, denoted as *LZ4* and *ZLIB*, respectively. Under the baseline scenario (i.e., all the on-storage pages use the conventional row-based format), the SSD with built-in transparent compression can achieve a compression ratio of 2.08. Compared with the baseline, *R2C-only*, *R2C-RLT*, *R2C-CLT*, *R2C-XOR* and *R2C-XOR-RLT* can improve the compression ratio by 20.6%, 30.5%, 44.9%, 37.5%, and 40.8%, respectively. Although the database mainly contains integers, compared with integer-page compression ratio shown in Fig. 8, the database compression ratios shown in Fig. 11 are noticeably higher and meanwhile the difference between the baseline and other scenarios is relatively smaller. This is mainly because not all the pages are filled up to $\frac{15}{16}$-full in the 160GB MySQL database. The less full one page is, the better the page can be compressed.

Finally, as shown in Fig. 11, MySQL's own lz4/zlib-based page compression achieves worse compression ratio than the SSD with built-in transparent compression. This can be explained as: When using its own page compression, MySQL stores each compressed 16KB page by using the *hole punching*

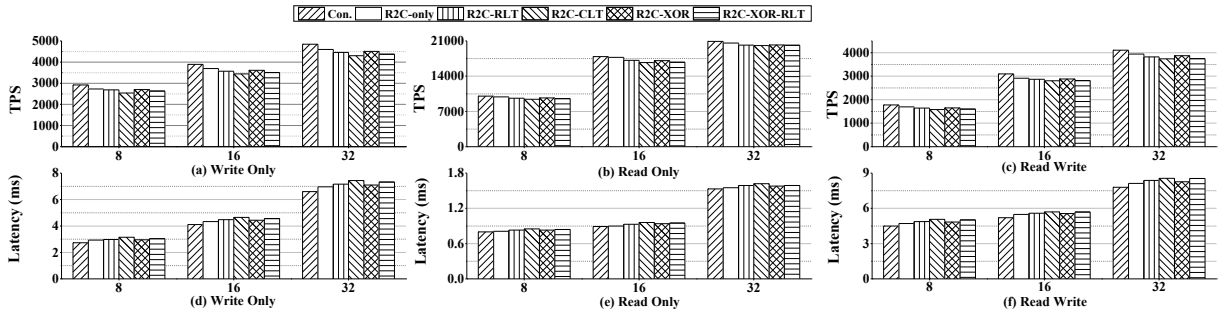

Fig. 10. Measured TPS and average latency when running MySQL with 8, 16, and 32 client threads under three different Sysbench OLTP workloads.

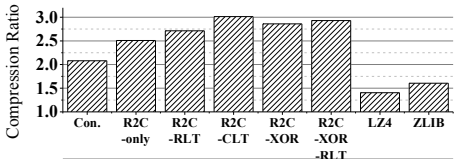

Fig. 11. Comparison of database compression ratio under the six different scenarios and MySQL's own lz4/zlib-based page compression.

of the underlying filesystem. As a result, each compressed page has to be 4KB-aligned on the storage device. For example, if MySQL's own page compression can compress a 16KB page to 9KB, the compressed page has to occupy 12KB space on the storage device. This leads to significant storage space under-utilization. However, the SSD with built-in transparent compression can tightly place all the compressed 4KB blocks together on the NAND flash memory storage media. Hence, although MySQL's own page compression compresses each 16KB entirely, it still cannot match the compression ratio achieved by the SSD with built-in transparent compression.

V. Related Work

Database research community has long studied the benefit and trade-off of applying compression to database (e.g., see [2], [16], [24], [8], [4]). A variety of compression techniques [11], [25], [6], [19] were developed based upon the principles of entropy coding [14], [26] and LZ-search [27], [28]. Few other compression algorithms (e.g., vector quantization [18]) have also been considered over the years. Early prior work primarily focused on relational database with row-based page format, and has led to the integration of compression schemes at the record/page (and even table) level in commercial relational database (e.g., DB2 [15] and Oracle [20]) and open-source relational database such as MySQL and MariaDB. By implementing an emulator for SSD with built-in transparent compression, Zuck *et al.* [31] studied the options of integrating transparent compression into SSD, and demonstrated its potential of reducing storage cost for relational database without sacrificing TPS performance.

VI. Conclusions

This paper presents a dual in-memory vs. on-storage page format design framework for relational database to take full advantage of modern storage hardware with built-in transparent compression. The key is to convert on-storage pages from conventional row-based format to more compressible column-based format. We complement row-column conversion with column data transformation to further improve the on-storage page compressibility. We present a set of light-weight data transformation schemes, and analyze how the compression algorithm being implemented by storage hardware could impact the effectiveness of different data transformation schemes. We modified the latest MySQL 8.0 to support the proposed dual page format, and experimental results on a commercial SSD with built-in transparent compression show that the proposed design approach can significantly improve the data compression ratio at almost negligible OLTP performance degradation.

VII. Acknowledgement

This work is supported by the National Science Foundation under Grant CNS-2006617.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 967–980, 2008.

[2] P. A. Alsberg. Space and time savings through large data base compression and dynamic restructuring. *Proceedings of the IEEE*, 63(8):1114–1122, 1975.

[3] AWS Graviton Processor. https://aws.amazon.com/ec2/graviton/.

[4] M. A. Bassiouni. Data compression in scientific and statistical databases. *IEEE Transactions on Software Engineering*, (10):1047–1058, 1985.

[5] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Proceedings of the Usenix Conference on File and Storage Technologies (FAST)*, volume 215, 2003.

[6] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 271–282, 2001.

[7] D. Chiou, E. Chung, and S. Carrie. (Cloud) Acceleration at Microsoft. *Tutorial at Hot Chips*, 2019.

[8] G. V. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[9] Dell EMC PowerMax. https://delltechnologies.com/.

[10] S. Deorowicz. Silesia compression corpus. *URL: http://sun. aei.polsl.pl/˜sdeor/index.php*, 2014.

[11] G. Graefe and L. Shapiro. Data compression and database performance. In *IEEE Symposium on Applied Computing*, pages 22–23, 1991.

[12] E. F. Haratsch. SSD with Compression: Implementation, Interface and Use Case. In *Flash Memory Summit*, 2019.
[13] HPE Nimble Storage. https://www.hpe.com/.
[14] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[15] B. R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB*, volume 94, pages 695–704, 1994.
[16] C. A. Lynch and E. B. Brownrigg. Application of data compression to a large bibliographic data base. In *Proceedings of the International Conference on Very Large Data Bases*, pages 435–447, 1981.
[17] LZ4. https://github.com/lz4/.
[18] W. K. Ng and C. V. Ravishankar. Relational database compression using augmented vector quantization. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 540–549. IEEE, 1995.
[19] S. O'connell and N. Winterbottom. Performing joins without decompression in a compressed database system. *ACM SIGMOD Record*, 32(1):6–11, 2003.
[20] M. Poess and D. Potapov. Data compression in oracle. In *Proceedings of VLDB Conference*, pages 937–947. Elsevier, 2003.
[21] Pure Storage FlashBlade. https://purestorage.com/.
[22] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.
[23] ScaleFlux Computational Storage. http://scaleflux.com.
[24] D. G. Severance. A practitioner's guide to data base compression tutorial. *Information Systems*, 8(1):51–62, 1983.
[25] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):55–67, 2000.
[26] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.
[27] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.
[28] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.
[29] zlib. http://zlib.net.
[30] Zstandrad (ZSTD). https://github.com/facebook/zstd.
[31] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik. Compression and SSDs: Where and how? In *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.