Leveraging Page-Level Compression in MySQL —a Practice at Baidu

Jingwei Ma[†§], Boxue Yin[§], Zhi Kong[§], Yuxiang Ma[†§], Chang Chen[†§],
Long Wang[§*], Gang Wang[†✉], Xiaoguang Liu[†✉]
[§]Baidu Inc., Beijing, China
[†]College of Computer and Control Engineering, Nankai University, Tianjin, China

***Abstract*—Facing large scale of data sets, disk I/O seems still one of the bottlenecks in DBMS. In the mean time, the CPU resource is not fully utilized. So compression is introduced to take use of the computing resource and largely reduces the storage overhead. Also, the commonly used compression algorithm can improve the performance when the database runs on HDD. With SSD, however, the performance for both read and write could be negatively affected by the slow process of compression and decompression. By quantitatively analyzing the impact of compression, we proposed a balanced compression solution on SSD, in which the read performance is accelerated by using a compression algorithm (lz4hc) with an extreme high decompression speed and an asynchronous compression mechanism is introduced to reduce the write latency by moving compression to the background. We test the performance on the real data set collected from the online database systems in Baidu. The results show the read performance on SSD is improved by 25% compared to the uncompressed database and 36% compared with commonly used zlib compression. Meanwhile, the write performance is up to 20% and 33% better than the synchronous compression on lz4hc and zlib.**

I. Introduction

Relational databases are widely used in internet applications and play a key role in the infrastructure for transaction computing and data storing. With the increasingly rapid growth of data and active transactions, disk I/O becomes the bottleneck in current large-scale database systems while the utilization of CPU is usually very low. This is a waste of ever-increasing computing power of modern CPUs. Compression can make use of those idle CPU cycles and bring benefits to DBMS in several aspects so that it is widely used in modern database systems [1], [2], [3], [4], [5], [6], [7]. It could considerably reduce the cost of storage, especially for systems that use SSD as the secondary storage medium. Besides, as the data is compressed, the main memory can hold a larger proportion of the entire data. This will result in a higher cache hit rate and improve the performance of databases running on slow storage medium. The performance improvement is mainly reflected on read latency.

Various compression techniques have been applied to different domains of databases. Dictionary-based compression [8] is perhaps the most prevalent compression technique in databases nowadays and can be used for any data type. Lots of commercial systems apply this technique at the page level [9] and a distinct dictionary is maintained for every page, which ensures that decompression does not need additional I/Os. Also, many studies have been concentrated on compressing column-oriented databases. Storing data in a column-major fashion presents a number of opportunities for improving performance from compression perspective compared to row-oriented architectures. In a column-oriented database, encoding multiple values together is easy because they are of the same attribute and are often quite similar to each other. As a consequence, compression ratio is generally higher in column-stores.

Besides the compression ratio, performance is also a very important metric for DBMS. Page-level compression is introduced to MySQL to improve the performance on HDD and reduce the consumption of costly SSD [10]. Since a compression failure can result in split of the page, which requires exclusive lock and will seriously impact the performance. Facebook introduces the *Adaptive Padding* technique to reduce the compression failure rate to only 5% and remarkably improve the write performance.

However, it is still a challenge to design a balanced solution that achieves both high read performance and high write throughput. Also, the compression algorithm zlib used in MySQL has never been proved to be the best one. Typically, compression algorithms are evaluated in terms of compression ratio, compression speed and decompression speed. Actually, there is no compression algorithm superior to others in all of these metrics. Algorithms with higher compression ratio (such as zlib [11] and lib7zip [12]) often have slow compression and decompression speed which is detrimental to the write and read performance. Meanwhile, algorithms with a higher compression speed (such as lzo [13] and snappy [14]) usually have a worse compression ratio which reduces the cache hit rate. Furthermore, the performance of compressed databases is not simply determined by these isolated compression metrics. The database performance itself is not even one-dimensional. These compression performance indicators and other factors together determine the overall database performance.

This study does not focus on compression performance indicators in isolation. Instead, we try to build an analytical model to reveal how these indicators and other factors together impact the performance of compressed database. Inspired by the analysis, we replace zlib algorithm commonly used in MySQL by lz4hc [15], whose decompression speed is 9 times faster than zlib on the test data set. So we can obtain a better read performance than non-compression even with fast storage

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.177

devices like SSDs. For write, by adopting an asynchronous compression mechanism, we effectively alleviate the impact of multiple compression on write performance.

We evaluate our compression scheme on real world database workload from Baidu, the largest Chinese search engine. The read performance on SSD can be improved by up to 36% compared with zlib. The write performance is improved by 20% when there is one insert thread.

The remainder of the paper is organized as follows. We describe the related studies in this area in Section II. Section III gives a description of compression in MySQL. The quantitative analytical model is presented in Section IV. We detail the optimization methods in Section V and evaluate them on real world workload in Section VI. Section VII summarizes the paper at last.

II. Related Work

The idea of compression in database [16], [17] is as old as the concept of database. Many of the early works concentrated on reducing the size of stored data. Most of the major traditional database vendors have offered compression technique in their product. Westmann *et al.* [2] have shown how compression can be integrated into a RDBMS. Poess and Potapov [1] presented Oracle's compression schema, which can enable a data warehouse to store several times more raw data without increasing the total disk storage or impacting query performance. Bhattacharjee *et, al.* [6] presented the design of index compression in DB2 LUW. The technique is able to compress index data efficiently with no performance penalty for query processing even with performance promotion for certain operations. These researches concentrate on the implementation of compression in database to reduce the storage space, while reasonable use of compression could bring performance improvement.

The efficiency of compression and decompression has also been considered in recent years. Super-scalar RAM-CPU cache compression [4] takes advantage of the characteristics of hardware to reduce the I/O bandwidth required in compressed database and information retrieval systems. Decompression is applied between RAM and CPU cache rather than between RAM and I/O, which allows database to store more compressed data in the main memory and gains high decompression speeds. Page-level compression is introduced to MySQL to improve the query performance and reduce the footprint of the storage resource. Facebook proposed the Adaptive Padding technique to reduce the compression failure rate [10]. With the failure rate reduced to only 5%, the write performance is dramatically improved. We find compression itself also affect the write performance seriously, so we design an asynchronous compression mechanism to move compression to the background.

Stonebraker *et al.* [3] designed C-Store which is a read-optimized compressed relational DBMS. Data is organized by column after being encoded. It can simultaneously achieve very high performance on warehouse-style queries and reasonable speed on OLTP-style transactions. Abadi *et al.* [5] extended C-Store with a compression sub-system. They showed that a significant performance improvement can be gained by implementing light-weight compression, and focusing on column-oriented compression allows the DBAs to operate directly on the compressed data. However, the performance of column-oriented compression relies heavily on the design of database. Databases, especially those created for analytic purposes, often show an extremely large degree of correlation among columns. Pararies *et al.* [18] developed an algorithm for detecting functional dependencies based on entropy measures which makes it possible to compress multiple logical columns together. Their experiments proved their algorithm performs well in scale with the number of columns and produces reliable dependence structure information. Muller *et al.* [19] proposed a compression manager automatically selecting the most appropriate dictionary format for every column in database. The adaptive compression can improve the overall performance by 10% using the same space or reduce the space consumption to 60% while maintaining the performance of the single format. Since MySQL InnoDB is row-based storage engine, we try to improve the performance of row-based page compression, which is more widely applicable.

Sushiula *et al.* [20] concluded various compression techniques for data stored in both row-oriented and column-oriented databases. Different techniques are applicable for different databases, while quantitative analysis is more difficult. Idreos *et al.* [9] introduced a method to estimate the size of an index if it is to be compressed. They analyzed the estimation accuracy for null suppression and dictionary compression. The estimator *SampleCF* draws an uniform random sample and returns the compression fraction on the sample. To our best knowledge, it is the first work to analyze the effectiveness of compression quantificationally.

III. Compression in MySQL InnoDB

Before presenting our analytical model and optimizing strategies, we introduce compression in MySQL InnoDB [10]. It is a row-oriented storage engine, in which the tables are structured as a clustered B^+ tree. The leaf page contains all columns of the table including the primary key and the internal node contains the index information only. Data is stored in the pages which are organized with the fixed size of 16KB.

With transparent compression, the page can be compressed into different fixed size page (1KB, 2KB, 4KB or 8KB) according to the configuration. As Figure 1 shows, the compression logic is embedded into the cache (buffer pool) module. When a query needs to acquire a data page, the system fetches the compressed page from disk to the buffer pool and decodes it before delivering it to user for processing. Since MySQL does not drop the compressed page after decompression, once accessed, both the compressed page and the corresponding uncompressed page reside in the buffer pool. So every uncompressed page has a corresponding compressed page in the buffer pool but a compressed page does not have to have a corresponding uncompressed page. The uncompressed pages take 10% of the total buffer pool size by default.

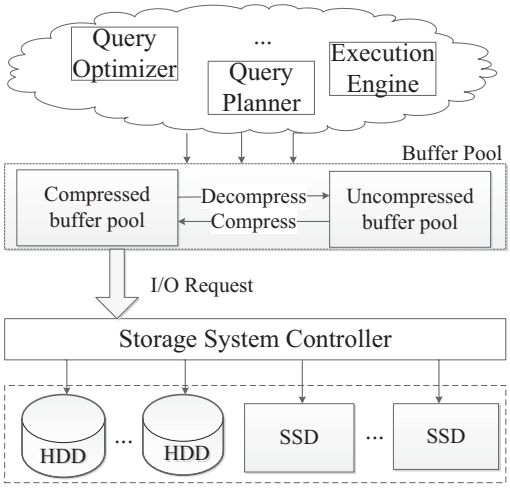

Fig. 1. Overview of Compression in MySQL

Figure 2 shows the compressed and uncompressed page structures. The compressed user records take a part of the compressed page. The rest is used to store the uncompressed information (such as page header, index and dense directory) and *modification log (mlog)* which is used to temporally store the modifications.

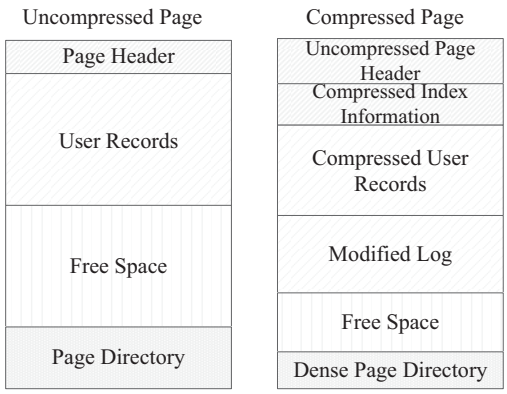

Fig. 2. Uncompressed page vs. Compressed page

A modification to the data table space, in fact operated on certain uncompressed page, needs to be reflected on the corresponding compressed page as well. When a new record comes, InnoDB attempts to insert the record into the uncompressed page at first. Once this operation has been successfully established, the record is inserted into the mlog area of the compressed page.

When the space for the *mlog* runs out, the system will do the compression. It takes the uncompressed page as the source instead of the mlog. As the records are continuously inserted into the compressed page, compression interval gets shorter, and eventually compression could fail due to the lack of space. When a failure occurs, the page will be split into two. A page split happens when lacking free space in the uncompressed page as well, which also happens in the configurations without compression. Split requires exclusive lock on the B^+ tree so that it may block other processes.

IV. Analysis

The performance of a database system is usually measured in terms of qps (queries per second) and ips (inserts per second) which closely relate to read and write latency in the storage engine. We quantitatively analyze relationship between system parameters and read and write performance.

A. Notations

We first define the notations used in our analytical model:

- PS_u: size of uncompressed page.
- PS_c: size of compressed page.
- T^r_{page}: page access time for read operation.
- BS: buffer pool size.
- D: size of data to be accessed.
- H_u: cache hit rate of uncompressed pages.
- H_c: cache hit rate of compressed pages.
- R: percentage of compressed pages in buffer pool.
- T^r_{disk}: disk access time for one read from disk.
- T_d: decompression time for one page.
- F_n: the size of free space after the n-th compression.
- r: compression ratio of the data.
- rec: the average record size.
- a_n: the data size needing to be compressed in the n_{th} compression.

B. Read Latency

Without compression, a page is directly accessed in the buffer pool or read from the disk. We ignore memory access time because the main memory access is several orders of magnitude faster than the disk access. Thus, the page access time for an uncompressed page can be expressed as the following equation.

$$T^r_{page} = T^r_{disk} \times (1 - H_u) \tag{1}$$

With compression on, the page access time can be regarded as the decompression time for the page if the compressed version of a page is hit in the buffer pool. If neither the uncompressed version nor the compressed version is found in the buffer pool, the compressed page must be read from the disk first then be decompressed. So the average page access time becomes:

$$\begin{aligned} T^r_{page} =& (T^r_{disk} + T_d) \times (1 - H_u - H_c) + T_d \times H_c \\ =& T^r_{disk} \times (1 - H_u - H_c) + T_d \times (1 - H_u) \end{aligned} \tag{2}$$

C. Cache Hit Rate for Leaf Nodes

The cache hit rate ascends in scale with the performance improvement. Though the data is constructed based on B^+ tree, we ignore the effect of internal nodes. This is because the internal nodes usually occupy small storage space and reside in buffer pool as they are frequently accessed. So internal nodes have little impact on the read performance.

The cache hit rate is determined by multiple factors including the cache size, the caching algorithm used and data access pattern. The latter two are various and their impact on cache hit rate is too complicated to be depicted accurately. However, the cache hit rate is positively correlated with the buffer pool size (in terms of the ratio of it to the total volume of accessed data). Thus, without compression we discuss the cache hit rate of leaf nodes by the following formula.

$$H_u \propto \frac{BS}{D} \tag{3}$$

With compression on, the size of memory taken by the uncompressed pages is $BS \times (1 - R)$. Hence the cache hit rate of leaf nodes in uncompressed area of buffer pool can be described by:

$$H_u \propto \frac{BS \times (1 - R)}{D} \tag{4}$$

Since every uncompressed page also has a corresponding compressed version in the buffer pool, these compressed pages take $(1 - R) \times r$ of the entire buffer pool. The compressed pages without uncompressed buddies take $R - (1 - R) \times r$ of the entire buffer pool. They contribute "compressed hits" and their original size is $\frac{(R-(1-R)\times r)\times BS}{r}$. So we have:

$$H_c \propto \frac{BS \times ((1 + r) \times R - r)}{D \times r} \tag{5}$$

D. Write Latency

Here we regard the compression ratio as a constant for a certain workload. During the $(i+1)$-th compression, the space taken by the i-th compressed data will not change. So the free space released by the $(i + 1)$-th compression is produced by the compression of *mlog* which is equal to F_i. Then we can get F_{i+1} from F_i as the following formula.

$$F_{i+1} = F_i \times (1 - r)$$

Initially, all the space in the compressed page is used as *mlog*. So $F_0 = PS_c$. Therefore, we can get any F_i.

$$F_i = (1 - r)^i \times PS_c$$

When the free space produced by compression cannot hold one compressed record, that is, $\frac{F_n}{r} < rec$, the page will be split. Thus, we can determine the number of compression operations for one page. Though the last compression will fail, it is also issued. So we take the ceiling.

$$n = \lceil \log_{1-r} \frac{rec \times r}{PS_c} \rceil$$

During the $(i + 1)$-th compression, besides the data that has been compressed in the first n steps, the data stored in the *mlog* area also needs to be compressed. Actually, the free space can be represented by F_i. However, we want to reveal the relationship of the amount of data needing to be compressed between two adjacent compression operations. So besides representing it by using F_i, we represent it using a_i, that is $(PS_c - r \times a_i)$. So we have the following formula to determine the data to be compressed in the $(i + 1)$-th compression.

$$a_{i+1} = a_i + (PS_c - r \times a_i) = PS_c + (1 - r)a_i$$

Assuming a notation t where:

$$a_{i+1} + t = (1 - r)(a_i + t)$$

We can solve t:

$$t = -\frac{PS_c}{r}$$

So:

$$a_n - \frac{PS_c}{r} = (1 - r)^n (a_0 - \frac{PS_c}{r})$$

Thus:

$$a_n = (1 - r)^n (a_0 - \frac{PS_c}{r}) + \frac{PS_c}{r}$$

The first compression happens when the compressed page is filled up for the first time. Thus $a_0 = PS_c$, so

$$a_n = (1 - r)^n (PS_c - \frac{PS_c}{r}) + \frac{PS_c}{r}$$

That is

$$a_n = (1 - r)^n \frac{(r - 1)PS_c}{r} + \frac{PS_c}{r}$$

The total volume of data compressed in n steps is:

$$\sum_{i=0}^{n} a_i = \frac{1 - (1 - r)^n}{r} \times PS_c + n \times \frac{PS_c}{r} \tag{6}$$

V. Optimization Methods

Based on the above analyses, we describe the problems and give our optimization methods in this section.

A. Read

According to Equation 2, there are following several ways to improve the read performance.

- Reduce disk access time.
- Improve the cache hit rate on the uncompressed pages.
- Reduce the decompression time.
- Pre-decompression.

For the first approach, the disk access time is an attribute that determined by the disk itself. A direct way is to change the disk to some kind of much more faster nonvalatile media. However, this raises the cost for the database. Taking the second way into consideration, the cache hit rate highly relies on the data size, the compression ratio and the data access pattern. Unfortunately, the database cannot determine the data users store in it and also cannot change the access pattern in which the user will access the data. We also consider asynchronous decompression in read, which is to decompress the page before it is really accessed. However, this method needs to know which page will be accessed in a short period of time. Without knowing the queries in advance, that seems impossible. Therefore, the most pratical way is to improve the decompression performance so that a compressed page can be quikly accessed. The compression algorithm used plays the key role in decompression speed.

B. Faster Decompression

Zlib is used in InnoDB by default. It gives a good compression ratio, which can benefit the cache hit rate. However, it also has a very low decompression speed. We use lz4hc compression algorithm to get a good read performance.

Some compression algorithms have a faster speed on both compression and decompression (lzo *etc*). However, they usually sacrifice some space saving to achieve this. On the contrary, lz4hc could achieve a very fast decompression speed while maintain a reasonable compression ratio. Though its compression speed is not fast enough to improve the write performance, we can put the compression tasks on background which will be presented in the next subsection.

To reveal how the decompression speed affect the read latency, we analyze the page access speed with different data-size/buffer-pool-size rate. If we refer the decompression speed in Table III and the random read throughput in Table II, get the cache hit rate by Formula 3 and 4, then we can get the average page access rate as data-size/buffer-pool-size rate varies from 4 to 32, which is shown in Figure 3.

When the rate is small, both zlib and lz4hc configurations can achieve much higher page access rate than that of the uncompressed configuration. However, as the rate grows, the gap among them gets narrow. Uncompressed configuration exceeds zlib configuration in an early stage. This is because the decompression speed of zlib is low compared with the read performance of SSD. On the contrary, configuration with lz4hc still keeps a higher performance. Overall, configuration with lz4hc always achieves the best performance, since its decompression speed is much faster compared with zlib decompression and the read performance of SSD.

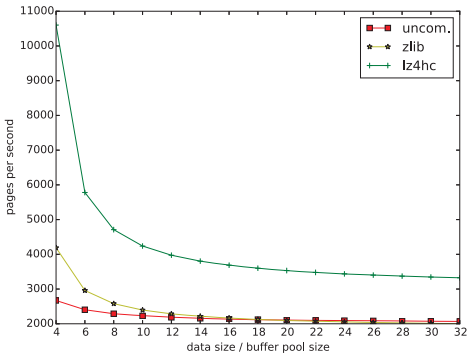

Fig. 3. Read Analysis

C. Write

Also, there are three ways to reduce the cost of writes.

- Improve the write performance of disk.
- Reduce the compression time.
- Pre-compression

As it is in read, the write performance of the disk relies on the attributes of the disk itself. Synchronous compression in database means some insert statements triggering compression must wait for the compression to be completed. This can cause a high latency for such write operations. There are compression algorithms with fast compression speed. However, the compression rate is not good.

So we try to bring the compression algorithms into the equations and find how they impact the write performance. If we use the lz4hc compression ratio and compression speed as in Table III, the time taken by compression and disk access is shown in Figure 4. When the size of record is small, compression can take a very long time due to many compression operations. As the size grows, the time of compression decreases since less compression operations are needed. However, even the record size equals to the compressed page size, the compression still takes much more time than the disk write.

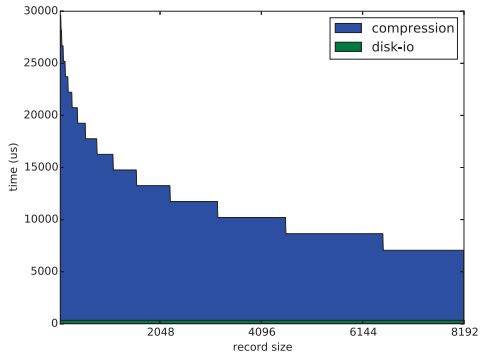

Fig. 4. Compression time as the record size varies using lz4hc

D. Proactive Asynchronous Compression

Compression only happens in the pages having recent inserts/updates. That indicates unlike decompression, the page needs to be compressed can be predicted. So we add a detection operation in the end of inserts/updates. Figure 5 shows the basic operations of the process. The system first builds several (16 by default) compression threads waiting in the thread-pool. When a page is almost full, the system wakes up a thread in the pool and arranges the page to that thread so that the page can be compressed in background. The waked up thread then tries to compress the page. In this way, the system can move the time-consuming compression to the background and reduce its overhead. To trigger the asynchronous compression, the system records the largest record size. After a successful insert, if the left space is less than the largest record size, the system will start an asynchronous compression.

Besides the asynchronous compression, we also propose advance split. When an insert operation needs to compress the page, the system will check if the page is just compressed. If so, the system just splits the page without trying to compress the page. Although this takes more space, it avoids a portion of compression failures.

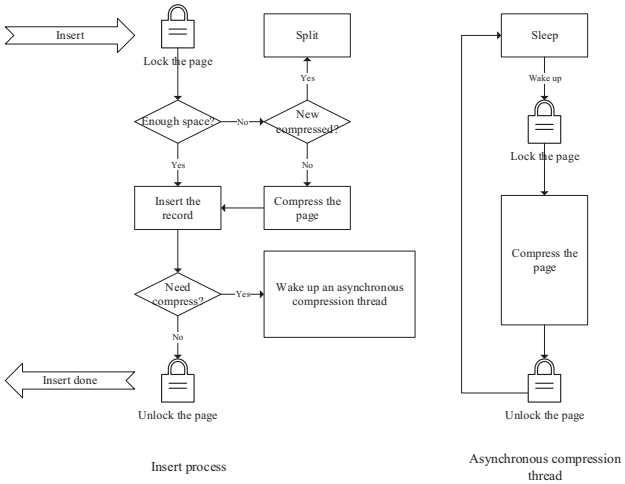

Fig. 5. Asynchronous Compression

TABLE I
PLATFORM

CPU	2 × INTEL Xeon E5-2620 6C 2.0GHZ
Memory	6 × 8G
HDD	2 × IBM SATA 500G 7200rpm
SSD	3 × IBM 600G MLC
OS	Centos4.3 2.6.32_1-15-0-0

VI. EVALUATION

The experiments are run on MySQL 5.6 with modified InnoDB. We focus on evaluating the selection and insertion performance, which represent the read and write efficiency, respectively. During the tests, we also collect the I/O utilization information in the above described operations.

"Zlib" indicates the system uses zlib as the compression algorithm and "lz4hc" represents the lz4hc compression algorithm is used in the system. During the insert tests, "async" means the asynchronous compression method and advance split are used and "sync" shows the system just uses the original synchronous compression process.

A. Experiment Setup

Table I lists the platform used in the tests. The operation system is installed on the HDD and the data is stored on the SSD. The data is collected from the on-line service in Baidu. It contains integers and texts, where the texts take most of the space (over 99%).

As the performance is seriously affected by the disk I/O, we first explore the data access features of the SSD with sysbench [21]. As the size of uncompressed page is 16KB and the target compressed page is 8KB, we set the block sizes to 16KB and 8KB. Table II illustrates the results. The SSD shows good performance on random read and sequential operations.

Table III shows the compression and decompression performance when the compression algorithms are directly used on the raw data. Although zlib can achieve a good compression rate, its speeds are slow both on compression and decompression. On the contrary, lz4hc can achieve a decompression speed of 586.45MB/s, which is 8.75 times faster than zlib. They have similar compression rate and compression speed.

TABLE II
DATA ACCESS PATTERNS OF SSD

Block Size	$8K$	$16K$
Random Read (MB/s)	25.73	31.30
Random Write (MB/s)	40.08	55.82
Sequential Read (MB/s)	58.22	53.60
Sequential Write (MB/s)	51.52	78.90

TABLE III
COMPRESSION & DECOMPRESSION ON RAW DATA

	com. (MB/s)	**decom. (MB/s)**	**com. rate**
zlib	14.85	67.04	0.24
lz4hc	17.57	586.45	0.30

We set the target page size as 8KB. Table IV shows the compression ratio using different configurations after inserting all the records. Compression leads to 50% space saving using zlib and lz4hc uses 55% of the original space to store the total data.

B. Read Performance

We first explore the read performance when replacing the original zlib compression algorithm with lz4hc. During the test, we set the buffer pool size to 36GB. The system selects 20% of all the data using the primary key.

Table V gives the average decompression time for a page using different zlib and lz4hc. As lz4hc has a faster decompression speed, the page decompression speed is 4.17 times faster than that of zlib.

We collect the cache hit ratio, which is shown in table VI. As zlib has a better compression ratio, it has a better cache hit ratio on compressed pages. The uncompressed pages take the same percentage of the entire pool and they have the same

TABLE IV
COMPRESSION RATIO IN DATABASE

	size(GB)	**compression ratio**
uncompressed	404	100%
zlib	202	50%
lz4hc	224	55%

TABLE V
AVERAGE PAGE DECOMPRESSION TIME

Algorithm	Time (us)
zlib	208.06
lz4hc	49.88

TABLE VI
CACHE HIT RATE

Algorithm	cache hit uncom	cache hit com
uncom	81.50%	–
lz4hc	72.54%	11.46%
zlib	72.46%	12.70%

TABLE VII
EXPECTED PAGE ACCESS TIME

Algorithm	page access time (us)
uncom	94.85
lz4hc	62.28
zlib	102.36

uncompressed page size. So they have the same cache hit rate on the uncompressed pages.

The page access time is established at the above cache hit rate according to Equation 1 and 2 in Table VII. Using zlib can results a longer page access time than the uncompressed one due to the slow decompression process. On the contrary, lz4hc can reduce the page access time as the decompression is extreme fast.

Figure 6 shows the qps (queries per second) when the number of connections varies from 1 to 16. In all the cases, lz4hc achieves the best performance as expected. And zlib gets the worst qps since it has a slow decompression process. As the number of connections grows, the qps increases almost linearly. The performance of lz4hc configuration is 20% better than that of the uncompressed configuration and 36% better than that of the zlib configuration with 1 connection. This improvement is not so large as in Table VII because other parts of the system also take a large portion of the entire time.

We also explore the disk access during the test. The results are illustrated in Figure 7. Since the page size is twice larger in the uncompressed configuration, it has the highest disk I/O. As lz4hc is faster than zlib, it deals with more pages from the disk. So the read I/O is higher than that of the zlib configuration.

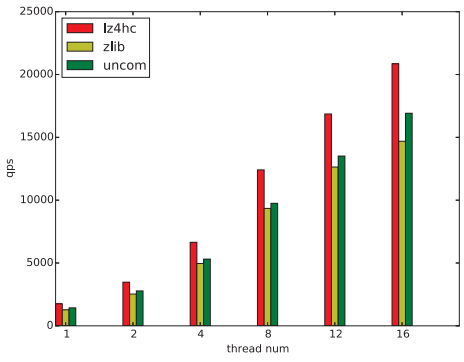

Fig. 6. QPS

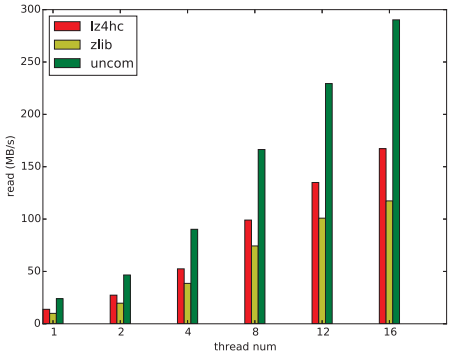

Fig. 7. Disk read throughput

C. Write Performance

During the write test, we use the records from the log files to test the ips (inserts per second) of the data base. The average record size is 612. The buffer pool size is set to 8GB. We insert 4×10^7 records for each test. Like read, we also test the performance with different threads. Since usually there are multiple tables in a database, each thread inserts records to 6 tables.

Figure 8 shows the ips when asynchronous compression is used in compression. As expected, moving compression to the background threads can reduce the waiting time for the insert process. Also, using compression can get a higher ips because compressed data pages take less disk I/O. The asynchronous compression performance is 20% and 33% better than the synchronous compression using lz4hc and zlib on one connection.

Also, lz4hc can get a higher ips than that of zlib. This is because it has a faster compression speed.

As the number of the threads increases, the ips grows until it reaches 8. The reason is that more threads issue more insert requests. However, after 8, more conflicts appear among those threads. The database system needs exclusive-locks to solve the conflicts, which affect the performance.

Figure 9 gives the information of the disk write throughput. Though uncompressed configuration consumes more disk bandwidth, it has less data pages written because its page size is twice the size of the compressed one. With asynchronous compression, more pages are written onto the disk. This is because the asynchronous compression compresses pages faster than the original synchronous compression.

Table VIII gives the results of the compression operations fraction for asynchronous compression and synchronous compression in the optimized system when the inserts are performed in one thread. Over 60% of the compression operations are done by the asynchronous compression threads. So the asynchronous compression can improve the write performance. Also, the advance split reduce the total compression operations by 2.4% for lz4hc and 3.8% for zlib (not shown in the table).

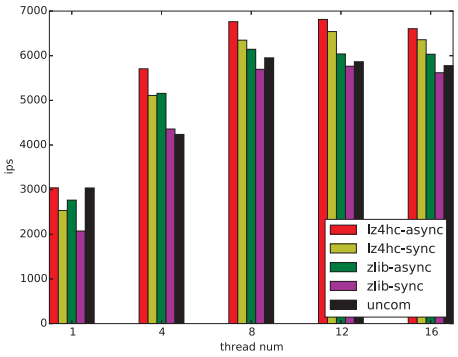

Fig. 8. IPS

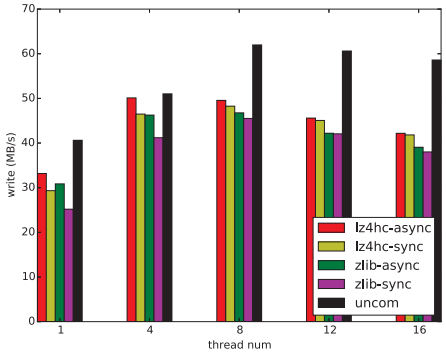

Fig. 9. Disk write throughput

VII. Conclusions

In this paper, we deeply analyze the page-level compression in DBMS. Based on these analyses, we use compression algorithm with fast decompression speed to accelerate the read performance of MySQL and employ asynchronous compression to improve the write performance. We evaluate our compression scheme on real world database workload from Baidu, the largest Chinese search engine. The read performance on SSD can be improved by up to 36% compared to zlib-based compression scheme. Meanwhile, the write performance can be improved by 33%.

Acknowledgments

This work is partially supported by NSF of China (grant numbers: 61373018, 11301288, 11550110491), Program for New Century Excellent Talents in University (grant number: NCET130301), the Fundamental Research Funds for the Central Universities (grant number: 65141021) and the Ph.D. Candidate Research Innovation Fund of Nankai University.

References

[1] M. Poess and D. Potapov, "Data compression in oracle," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 937–947.

TABLE VIII
Compression operation fraction (async. vs. sync.).

	async.	sync.
lz4hc	61.52%	38.48%
zlib	63.55%	36.45%

[2] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *ACM Sigmod Record*, vol. 29, no. 3, pp. 55–67, 2000.
[3] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 553–564.
[4] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 59–59.
[5] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 671–682.
[6] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat, "Efficient index compression in db2 luw," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1462–1473, 2009.
[7] L. Ki-Hoon, "Performance improvement of database compression for oltp workloads," *IEICE TRANSACTIONS on Information and Systems*, vol. 97, no. 4, pp. 976–980, 2014.
[8] D. J. Abadi, "Query execution in column-oriented database systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2008.
[9] S. Idreos, R. Kaushik, V. Narasayya, and R. Ramamurthy, "Estimating the compression fraction of an index using sampling," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 441–444.
[10] N.Ordulu and J.Tolmer, "Innodb compressiom: Present and future," in *Percona Live MySQL Conference*, 2013.
[11] zlib, *http://zlib.net/*.
[12] lib7zip, *https://code.google.com/p/lib7zip/*.
[13] lzo, *http://www.oberhumer.com/opensource/lzo/*.
[14] snappy, *https://code.google.com/p/snappy/*.
[15] lz4, *https://code.google.com/p/lz4/*.
[16] G. V. Cormack, "Data compression on a database system," *Communications of the ACM*, vol. 28, no. 12, pp. 1336–1342, 1985.
[17] B. R. Iyer and D. Wilhite, "Data compression support in databases," in *VLDB*, vol. 94, 1994, pp. 695–704.
[18] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krueger, "How to juggle columns: an entropy-based approach for table compression," in *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*. ACM, 2010, pp. 205–215.
[19] I. Müller, C. Ratsch, and F. Faerber, "Adaptive string dictionary compression in in-memory column-store database systems." in *EDBT*, 2014, pp. 283–294.
[20] S. Aghav, "Database compression techniques for performance optimization," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 6. IEEE, 2010, pp. V6–714.
[21] sysbench, *https://launchpad.net/sysbench/*.