SUMMARY  As data volumes explode, data storage costs become a large fraction of total IT costs. We can reduce the costs substantially by using compression. However, it is generally known that database compression is not suitable for write-intensive workloads. In this paper, we provide a comprehensive solution to improve the performance of compressed databases for write-intensive OLTP workloads. We find that storing data too densely in compressed pages incurs many future page splits, which require exclusive locks. In order to avoid lock contention, we reduce page splits by sacrificing a couple of percent of space savings. We reserve enough space in each compressed page for future updates of records and prevent page merges that are prone to incur page splits in the near future. The experimental results using TPC-C benchmark and MySQL/InnoDB show that our method gives 1.5 times higher throughput with 33% space savings compared with the uncompressed counterpart and 1.8 times higher throughput with only 1% more space compared with the state-of-the-art compression method developed by Facebook.

key words: database compression, performance, online transaction processing

1. Introduction

Due to the data explosion in recent years [4], data storage has become an important cost factor for large-scale database applications. Database compression can reduce data storage costs substantially, and also result in fewer I/Os needed to access the data. Database compression works well for workloads dominated by reads, but could show low performance for write-intensive workloads [5], [7], [10].

In this paper, we propose a comprehensive solution to improve the performance of compressed databases for online transaction processing (OLTP) workloads, which are write-intensive. We find that lock contention is a major performance bottleneck and avoid the lock contention by reducing page splits. We reserve enough space in each compressed page for future updates of records so that updates do not incur page splits in most cases. We also prevent page merges that make pages too dense and result in splits soon. We implement our solution in MySQL [5], which is the world’s most widely used open-source DBMS. We perform a detailed experimental evaluation on the performance impact of our solution using the TPC-C benchmark [9]. The experimental results show that our method gives 1.5 times higher throughput with 33% space savings compared with the uncompressed counterpart. Compared with the state-of-the-art compressed method [6], our method shows 1.8 times higher throughput with only 1% space overhead.

The rest of this paper is organized as follows. Section 2 introduces MySQL/InnoDB compression, and Sect. 3 reviews existing work. Section 4 presents our solution to reduce the lock contention of compressed database for OLTP workloads. Section 5 conducts experiments to evaluate the performance impact of our solution. Finally, Sect. 6 presents our conclusions.

2. MySQL/InnoDB Compression

MySQL/InnoDB tables are structured as a clustered B+-tree index where each record in the leaf page contains all columns of the table including the primary key. Secondary indexes in InnoDB are also B+-trees.

A table created with the compression option can use a smaller page size on disk than the usual 16KB default. A table and all of its indexes use the same compressed page size, and InnoDB compresses each page using zlib [11]. The compressed page size could be set to 1KB, 2KB, 4KB, 8KB, or 16KB. The optimal setting of the compressed page size depends on the type and distribution of data.

To access the data in a compressed page, InnoDB should decompress the page to its original 16KB form. If the compressed page is actively used, the buffer pool has the uncompressed version of the page as well. When both compressed and uncompressed images are present in the buffer pool, changes are applied to both to keep in sync.

InnoDB reduces recompression of pages by using the free space in the compressed page if available, which is called modification log (mlog). We insert records into this mlog in uncompressed form. When the space for the mlog runs out, we reorganize and recompress the page, resulting in smaller empty mlog. For updates, we insert the updated record into the mlog. The old version of the record is removed from the compressed page when the page is reorganized and recompressed. As we continuously insert records into the compressed page, recompression interval is getting shorter, and eventually recompression could fail. If it fails (a situation known as a compression failure), we split the page. We note that a page split also could happen when we lack free space in the uncompressed page. This happens when the page is compressed well, and the uncompressed page gets full before the compressed page does. This naive InnoDB compression method can incur many page splits for updates due to lack of free space in compressed pages as
will be explained in Sect. 4.

There are two ways to insert, update, and delete a record: optimistic and pessimistic. Optimism means that only in-page operation will be needed. In contrast, when the B*-tree index structure is supposed to be changed by splitting or merging pages, we perform pessimistic operations by locking the whole index in exclusive mode, which lead to low concurrency. For example, suppose that we want update a record in a page. We first try optimistic update, which fails if we cannot insert the updated version of the record into the mlog even after we reorganize and recompress the page. We then perform pessimistic update, which locks the index exclusively because page splitting may be needed.

3. Related Work

Roth and Horn [8] have provided a survey on lossless data compression techniques. Westmann et al. [10] have shown how compression can be integrated into a RDBMS. Poess and Potapov [7] have presented compression techniques used in Oracle for data warehouses and OLAP systems where the majority of the accesses are read only. Bhattacharjee et al. [2] have introduced index compression schemes used in DB2. Aghav [1] has introduced various compression techniques for difference domains of databases.

Recently, the engineers at Facebook [6] have proposed a method called adaptive padding to reduce the overhead of MySQL/InnoDB compression for insert-intensive workloads by avoiding compression failures. Compression failures are expensive in terms of CPU because they go through reorganization, recompression, failure, and finally page splitting. Here, reorganization and recompression spend precious CPU cycles for nothing. The method restricts the amount of data on the uncompressed page to keep it compressible, i.e., it tries to maintain the uncompressed page at a level that would nearly always compress into the compressed page size. When compression failure rate is higher than a threshold, it packs the uncompressed page less densely by adding unused space called pad to the uncompressed page. Because the uncompressed page gets full quickly due to the pad, a page split happens early before the compression failure. When the failure rate is fairly low, the method keeps removing the pad. This method focuses only on consecutive inserts of records and does not consider updates of records, which are common in OLTP workloads. It can incur many page splits for updates due to lack of free space in compressed pages as explained in the next section.

4. Our Solution for OLTP Workloads

The naive MySQL/InnoDB compression method and the Facebook one incur many page splits when we update records because they do not reserve any free space for future updates. After consecutive inserts of records, the methods would have very dense compressed pages with almost no free space available. Now, suppose that we update a record, and the update increases the compressed size of the record. Since the compressed page is already almost full before updating, a compression failure likely happens due to the increased size of the record. Thus, we should perform pessimistic update with exclusive lock because a page split can occur.

An update of a record will not incur a page split in general if enough free space for the update is available in the compressed page. To save space, we limit the size of the reserved free space to the largest record size of each table. Suppose that we update two records \( r, s \) successively. Since we have enough free space, we can insert the updated version \( r_u \) of \( r \) into the compressed page. We now try to insert the updated version \( s_u \) of \( s \), but there could be not enough free space left. We then reorganize and recompress the compressed page, resulting in replacing the old version \( r_o \) of \( r \) by the new one \( r_u \). If the compressed size \( |r_u^c| \) of \( r_u \) is smaller than or similar to \( |r_o^c| \) of \( r_o \), the free space for one record \( |s_o| \) is available again. This shows that if we have reserved enough free space for one record initially, the free space for one record is usually available again after updates. However, in the worst case where \(|r_u^c|\) is much larger than \(|r_o^c|\), the free space could be smaller than one record size, and we need to perform pessimistic updates.

We propose a novel method that reserves enough free space for one record in the compressed page. Algorithm 1 shows the part of the optimistic insert procedure modified to reserve the free space. Table 1 shows the summary of notations used in algorithms. Obviously, we fail to insert if the uncompressed page is full (Lines 1–3). This happens when the uncompressed page gets full before the compressed page does. There is enough free space left in the compressed page. In Lines 4–6, we reserve the space in a compressed page by limiting the number of records that can be stored in the page. If the number of records \( P_{n,pvc} \) on a page \( P \) is larger than a threshold, we return FAIL without insertion. We initially set the threshold to the maximum integer value and dynamically decrease it. Each table has own threshold. Since the updated record size in the future can be larger than the current record size, we use the maximum record size to ensure enough space. In Lines 7–13, if the mlog runs out, we reorganize and recompress the page. We then check if there will be enough space left after insertion. If not, we adjust the threshold and return FAIL without insertion (Lines 9–12). If optimistic insert fails, we use pessimistic insert, which locks the whole index in exclusive mode, splits the page, and inserts the record to an appropriate page.

We can also reduce page splits by preventing

| Symbols | Definitions |
|---------|-------------|
| \( P_{n,pvc} \) | the number of records on a page \( P \) |
| \( P_{free} \) | the amount of free space on a page \( P \) |
| \( P_{used} \) | the amount of used space on a page \( P \) |
| \( P^u \) | the uncompressed version of a page \( P \) |
| \( P^c \) | the compressed version of a page \( P \) |
| \( rec_{size} \) | the size of record to be inserted |
merges that make pages too dense. We modify the `btr_can_merge_with_page` procedure of InnoDB that checks if given two pages can be merged. Algorithm 2 shows the modified part of the procedure. In Lines 1~3, we do not merge if the merged page will lack free space in the uncompressed page. In the worst case where insertion will never happen to the merged page, we just waste space, but the case is rare for OLTP workloads. In Lines 4~6, we do not merge if the number of records in the merged page will be larger than the threshold obtained from Algorithm 1.

Algorithm 1 optimistic_insert
\begin{verbatim}
Input: a record to be inserted into a page \( P \)
Output: FAIL or SUCCESS
1. if \( |P_{\text{free}}^f| < \text{rec.size} \) then
2. return FAIL
3. end if
4. if \( P_{\text{new}} > \text{threshold} \) then
5. return FAIL
6. end if
7. if \( |P_{\text{free}}^f| < \text{max.rec.size} \) then
8. reorganize and recompress the page
9. if \( |P_{\text{free}}^f| - \text{rec.size} < \text{max.rec.size} \) then
10. threshold ← \( P_{\text{new}} - 1 \)
11. return FAIL
12. end if
13. end if
14. insert the record into \( P_{\text{free}}^f \) and \( P_{\text{free}}^f \)
15. return SUCCESS
\end{verbatim}

Algorithm 2 btr_can_merge_with_page
\begin{verbatim}
Input: two pages \( P \) and \( Q \) where \( |P_{\text{free}}^f| > |Q_{\text{free}}^f| \)
Output: TRUE or FAILED
1. if \( |P_{\text{free}}^f| - |Q_{\text{free}}^f| < \text{max.rec.size} \) then
2. return FALSE
3. end if
4. if \( P_{\text{new}} + Q_{\text{new}} > \text{threshold} \) then
5. return FALSE
6. end if
\end{verbatim}

5. Experimental Evaluation

5.1 Experimental Setup

We investigate the performance impact of our solution using TPC-C [9], which is a standard benchmark simulating real OLTP workloads. The workload of TPC-C is composed of two read-only and three write-intensive transactions, which together provide many concurrent random reads and writes to the storage device. We use DBT-2 [3], which is an open-source implementation of the TPC-C specification. We use 20 database connections, 10 terminals per warehouse, and the duration of two hours. To measure the maximum performance, we set the key and thinking time to zero. Unless specially stated, we use a large scale factor of 1000 (about 100GB) because compression is practically useful when the database is large. In Sect. 5.2.5, we use a small scale factor of 10 (about 1GB) to test the worst case for compressed methods where the database is small enough to fit in the buffer pool. We use the TPC-C standard mix of transactions, except in Sect. 5.2.6 where we conduct experiments by varying the ratios of read-only and write-intensive transactions to evaluate the effect of the ratios of read I/O and write I/O.

We implement all the algorithms in MySQL Community Server 5.6.13 and run DBT-2 on top of it. The important parameters of MySQL are shown in Table 2. To minimize the interference by data caching at the OS layer, we use direct I/O (0_DIRECT). We set `innodb_log_compressed_pages` to OFF because we are not going to attempt to recover from a crash using a different version of zlib. We use default compressed page size 8KB and default compression level 6. To assure a controlled setting, we fix the ratio between the numbers of uncompressed pages and compressed pages in the buffer pool to 1:8. Finding the optimal setting of the ratio is not a focus of this paper and is left as a further study.

We compare our method, the method [6] developed by Facebook (shortly, the FB method), and the original MySQL/InnoDB compression method (shortly, the naive method). We measure space savings, loading performance, and transaction throughput. We define the space savings (SS) according to literature [7], [8].

\[
SS = \left( \frac{\text{uncompressed size} - \text{compressed size}}{\text{uncompressed size}} \right) \times 100\% \tag{1}
\]

All the experiments have been conducted on a Linux PC with an Intel Core i7-2600K CPU and 4GB of main memory. Unless specially stated, we use a Samsung 840 Series 256GB SSD because SSDs are more expensive than HDDs in terms of price per GB, and thus, compression is more important for them. In Sect. 5.2.7, we conduct experiments using a Western Digital 1TB HDD.

5.2 Experimental Results

5.2.1 Space Savings

Table 3 shows the database sizes of each method after running TPC-C benchmark. Our method reduces the database size from 97.59GB to 65.84GB yielding a space saving of 32.54%. Compared with the FB method, our method use 1.23% more space because we reserve space in compressed pages, but the space overhead is very small.
Table 3  Compression results of TPC-C database.

|                | No Comp. | Our Method | FB Method | Naive Method |
|----------------|----------|------------|-----------|--------------|
| DB size (GB)   | 97.59    | 65.84      | 65.04     | 64.92        |
| space savings (%) | N/A      | 32.54      | 33.36     | 33.47        |

Table 4  Loading performance.

|                | No Comp. | Our Method | FB Method | Naive Method |
|----------------|----------|------------|-----------|--------------|
| loading time (s) | 5.15     | 8.63       | 11.73     | 16.72        |
| comp. time (s)  | N/A      | 3.16       | 6.06      | 9.912        |
| CPU usage (%)   | 33       | 52         | 56        | 55           |
| # comp. ops (×10⁶) | N/A    | 2.9        | 3.8       | 5.4          |
| # split ops (×10⁶) | 5.8     | 7.9        | 7.3       | 7.1          |
| comp. failure (%) | N/A    | 0.0        | 2.5       | 20.6         |

Table 5  The percentage of dense compressed pages.

|                | No Comp. | Our Method | FB Method | Naive Method |
|----------------|----------|------------|-----------|--------------|
|                | N/A      | 0%         | 72%       | 77%          |

5.2.2  Loading Performance

Table 4 shows the database loading performance of each method. We observe that the loading times tend to be 1.67 to 3.23 times longer for compressed methods because loading with compression is a CPU-bound operation. The FB method is 1.42 times faster than the naive method because it reduces recompression operations by splitting pages early before compression failures. Our method is 1.36 times faster than the FB method because we split pages earlier to reserve free space.

We note that when we sequentially load tables, page splits do not incur lock waits. However, when we run multiple transactions concurrently, page splits incur a lot of lock waits as we can see in the subsequent sections.

5.2.3  The Percentage of Dense Compressed Pages

Table 5 shows the percentage of dense compressed pages after database loading. We define dense compressed pages as compressed pages whose free space is smaller than the average size of records of each table. The FB method and the naive method lack free space for more than 70% of compressed pages and likely perform pessimistic updates for those pages. In contrast, our method reserves enough free space for all compressed pages so that pessimistic updates do not happen in most cases. The FB method has a slightly less number of dense compressed pages than the naive method because it splits pages earlier to avoid compression failures.

5.2.4  Transaction Throughput for a Large Database

Figure 1 shows transaction throughput measured in transactions per minute (TPM) for a scale factor (SF) of 1000 where the database size is much larger than the buffer pool size. Our method shows 1.45 times higher TPM than the uncompressed method, 1.76 times than the FB method, and 1.91 times than the naive method.

Table 6  Average elapsed time per transaction.

|                | No Comp. | Our Method | FB Method | Naive Method |
|----------------|----------|------------|-----------|--------------|
| Transaction    | 271      | 185        | 339       | 367          |
| Lock Wait      | 86       | 49         | 243       | 257          |
| Read I/O       | 164      | 113        | 45        | 49           |
| Comp. + Decomp.| 0        | 6          | 9         | 11           |

Table 7  Average number of page splits per transaction.

|                | No Comp. | Our Method | FB Method | Naive Method |
|----------------|----------|------------|-----------|--------------|
|                | 0.28     | 0.16       | 0.38      | 0.47         |

Table 8  Average I/O queue length and request size.

|                | No Comp. | Our Method | FB Method | Naive Method |
|----------------|----------|------------|-----------|--------------|
| Queue Length   | 6.67     | 6.95       | 3.22      | 3.21         |
| Size (KB)      | 19.90    | 12.40      | 15.07     | 15.22        |

5.2.5  Transaction Throughput for a Small Database

Figure 2 shows transaction throughput for SF 10 where the
Database compression can significantly reduce storage costs, but existing methods show low performance for OLTP workloads. We have significantly improved the performance of compressed databases for OLTP workloads by avoiding lock contention. We have shown that our method greatly reduces page splits, resulting in better concurrency. Our work makes database compression a viable solution for OLTP workloads.

Acknowledgements

The present Research has been conducted by the Research Grant of Kwangwoon University in 2013.

References

[1] S. Aghav, “Database compression techniques for performance optimization,” Proc. Int’l Conf. Computer Engineering and Technology (ICCET), pp.714–717, April 2010.
[2] B. Bhattacharje, et al., “Efficient index compression in DB2 LUW,” Proc. Int’l Conf. Very Large Data Bases (VLDB), pp.1462–1473, Aug. 2009.
[3] DB2 Benchmark Tool, http://dev.mysql.com/downloads/benchmarks.html
[4] S. Lee and I. Lee, “A secure index management scheme for providing data sharing in cloud storage,” J. Information Processing Systems, vol.9, no.2, pp.267–300, June 2013.
[5] MySQL 5.6 Reference Manual, http://dev.mysql.com/doc/refman/5.6/en/
[6] N. Ordulu and J. Tolmer, “InnoDB compression: Present and future,” Percona Live MySQL Conference, April 2013.
[7] M. Poess and D. Potapov, “Data compression in oracle,” Proc. Int’l Conf. Very Large Data Bases (VLDB), pp.937–947, Sept. 2003.
[8] M.A. Roth and S.J.V. Horn, “Database compression,” ACM SIGMOD Record, vol.22, no.3, pp.31–39, Sept. 1993.
[9] Transaction Processing Performance Council (TPC), TPC BENCHMARK C, Standard Specification, Revision 5.11, Feb. 2010.
[10] 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, Sept. 2000.
[11] zlib, http://zlib.net