NVM aware MariaDB database system

Conference Paper · August 2015

DOI: 10.1109/NVMSA.2015.7304362

CITATIONS
2

READS
589

5 authors, including:

Jan Lindström
MariaDB Corporation
34 PUBLICATIONS **234** CITATIONS
SEE PROFILE

Dhananjoy Das
SanDisk
5 PUBLICATIONS **25** CITATIONS
SEE PROFILE

Dulcardo Arteaga
Florida International University
11 PUBLICATIONS **75** CITATIONS
SEE PROFILE

Nisha Talagala
Fusion-io
27 PUBLICATIONS **339** CITATIONS
SEE PROFILE

NVM Aware MariaDB Database System

Jan Lindström*, Dhananjoy Das†, Torben Mathiasen†,Dulcardo Arteaga†, Nisha Talagala†

*MariaDB Corporation
†SanDisk Corporation

***Abstract*—MariaDB is a community-developed fork of the MySQL relational database management system and originally designed and implemented to use traditional spinning disk architecture. Now that devices with Non-Volatile memory (NVM) technologies are available, MariaDB addresses this challenge by adding support for NVM devices and introduces NVM Compression method. NVM Compression is a novel hybrid technique that combines application level compression with flash awareness for optimal performance and storage efficiency. Utilizing new interface primitives exported by Flash Translation Layers (FTLs), we leverage the garbage collection available in flash devices to optimize the capacity management required by compression systems. We implement NVM Compression in the popular MariaDB database and use variants of commonly available POSIX file system interfaces to provide the extended FTL capabilities to the user space application. The experimental results show that the hybrid approach of NVM Compression can improve compression performance by 2-7x, deliver compression performance for flash devices that is within 5% of uncompressed performance, improves storage efficiency by 19% over legacy Row-Compression, reduce data writes by up to 4x when combined with other flash aware techniques such as Atomic Writes, and deliver further advantages in power efficiency and CPU utilization.**

I. Introduction

Traditionally compression has been extensively used to save expensive resources of capacity and bandwidth. In Non-volatile memory devices Flash Translation Layer (FTL) is an matching location to perform compression, because it already performs sophisticated data management [10], [17], [24]. A Flash Translation Layer is responsible for the mapping of Logical Block Address (LBA) updates to physical block updates (PBA) and thus can concurrently perform the compression. Flash has quite a high cost, lower capacity compared to traditional disk drives, and lower write endurance [7], [23]. These make compression even more attractive since it can reduce the write amplification, and increase the endurance for flash devices. Furthermore, FTL can provide compression functionality transparently to applications [24]. Integrating compression to FTL may be ideal for flash, but benefits of application level compression originating from better knowledge of application patterns can be lost. Compression has been earlier successfully used e.g. on key-value stores [1], [11]. Furthermore, non-volatile memory device drivers and file systems have started to appear [12]–[14].

Previous research on non-volatile memory storages have been concentrated on specific sub-systems of database architecture like checkpointing [8], combining main-memory and non-volatile memory [9], [14], [22], page size selection [18] or buffer replacement algorithm [6].

In this paper we propose a hybrid design where applications can have a control of compression techniques, while gaining some of the benefits that can come from integrating with an FTL for Flash Aware Compression. We implement such a hybrid design in the context of the MariaDB/MySQL open source database. MySQL legacy compression was designed for the traditional disk drives, thus the implementation seems outdated on modern storage devices and hardware. Current MySQL implementations is both complex and its performance compared to uncompressed tables is poor leaving many users concerned on robustness and usability of this feature.

The paper makes the following contributions:

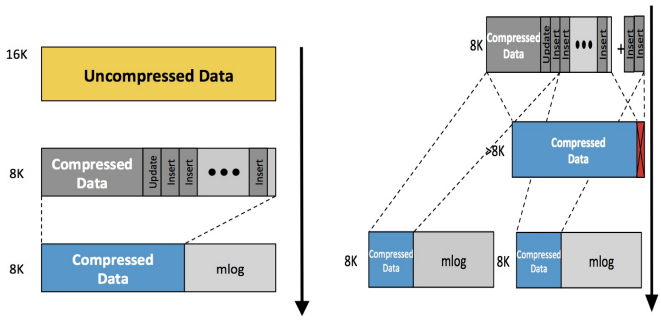
- Describes the design and implementation of a novel compression mechanism that combines the advantages of application level awareness and FTL integration.
- Describes ways to integrate such an approach into an operating system stack through a combination of new interface primitives and file system support.
- We include performance study showing that this approach can improve compression performance by 2-3x, that is within 5% of uncompressed performance, improve storage efficiency by 19%, reduce data writes by 4x using Atomic Writes.

II. Background

MySQL supports various different storage-engine options. InnoDB and XtraDB are the two most popular storage engines used by the MySQL community today. MariaDB [1] [3] is a community-developed fork of the MySQL relational database management system. The intent is to maintain high compatibility with MySQL, ensuring a ”drop-in” replacement capability with library binary equivalence and exact matching with MySQL APIs and commands. In InnoDB storage engine, data records are stored in files with pre-configured page size units, default at 16KB per page. As shown in Figure 1(a), row compression compresses data rows into a predefined compressed data block, size defined by an admin command at database table-create time. Each of the compressed blocks contains compressed data and a modify log-region (mlog) where further block changes are logged. As the table content changes, deltas are appended to the mlog until it runs out of space, at which point the compressed data is de-compressed, the mlog entries applied, and the resulting data-block is re-compressed.

This opens up the possibility of Compression Insert Failure, where the newly compressed block is too big to fit in the predefined fixed compressed block size. This failure leads to a node (page) split where attempts are made to fit this block

[1] www.mariadb.org



(a) MySQL Row Compression. (b) Insert Failure

Fig. 1. Traditional MySQL Row Compression.

into an alternate location in the block map, with the need to re-balanced the tree until all the data is successfully inserted. This can lead to a need to re-balance the block map. Figure 1(b) shows the steps involved to handle a compression insert failure. First, an adjacent page needs to be read and decompressed. Entries are then inserted into these pages and the allocation maps are re-balanced until all the data is successfully inserted.

Depending on the workload and the frequency of insert failures, performance may suffer since insert failures consume CPU cycles and add other data management overhead. This complexity leads to much of the performance issues seen today in MySQL Row-Compression deployments. Substantial work has been done in the last several years to attempt to solve this problem [4], [5], [15], [20].

III. NVM Compression

The goals of our compression solution, NVM Compression, are to deliver high speed and high efficiency flash compression that combines the benefits of application level data knowledge and the FTL's awareness and operation of management. FTLs benefit from the inherent presence of an indirection map that translates logical addresses to physical addresses, and the existing high performance garbage collection that coalesces available free space. To combine these benefits, we create an architecture where the compression is performed at the application level while the free space and the compressed data block are managed at the FTL level.

NVM Compression design has three elements:

- The primitives exported by the FTL that enable access to its thin-provisioning.
- Export primitives through NVMFS file system [21].
- Use of the primitives in an application level compression engine.

The NVMFS [21] POSIX compliant file system re-exports the functionality as file level interfaces as described in Table I.

MariaDB offers application developers a way to specify which tables are stored on a file system supporting atomic writes:

```
1: create table atomictable (x int unsigned not null primary
   key)
2: engine = innodb
3: data directory = '/mnt/fusionio/data';
```

Primitives	Details
Sparse	Sparse address space allows consuming applications to offload data allocation management to the FTL.
PTRIMs	Persistent TRIMs enable guaranteed deletes of a data block at a given virtual address.
Atomics	Atomic-write guarantees that no part of the buffer will be partially written.

TABLE I. NVM Primitives

Atomic Writes issued by the application are passed by NVMFS directly into the FTL as atomic operations. Effects of atomic writes have been researched more thoroughly in [16].

Instead of storing both compressed and uncompressed pages on buffer pool, we store only a uncompressed 16KB pages in buffer pool. This avoids very complex logic on when a page needs to be re-compressed or when to add a change to mlog. Similarly, there is no need to do page splits etc. Before creating a page compressed table, make sure the innodb_file_per_table configuration option is enabled, and innodb_file_format is set to Barracuda.

MariaDB provides also a way to define tables that should use page compression, what compression level is used if available by used compression algorithm.

```
1: SET GLOBAL innodb_compression_algorithm = lz4;
2: create table atomictable (x int unsigned not null primary
   key)
3: engine = innodb
4: page_compressed = ON page_compression_level = 6
5: data directory = '/mnt/fusionio/data';
```

All these are stored to InnoDB data dictionary persistently and all values can be changed with normal alter table SQL-clause. If compression level or algorithm is not specified, a system global value is used. Compression level can be specified using innodb_compression_level and algorithm with innodb_compression_method configuration variables. Both of these can be changed dynamically without server shutdown.

When a page is modified, it is compressed just before it is written and only a compressed size (aligned to sector boundary) is written (see Figure 2). If compression fails because of compression failure we write uncompressed page to the file space. The compressed page is stored with the resulting size in the virtual address allocated to the uncompressed block and a PTRIM operation is issued for the remainder of the address range (using fallocate(PUNCH HOLE)) to inform NVMFS and the FTL that the remainder of the virtual addresses are now empty and can be garbage collected as appropriate.

This has been implemented with special page type FIL_PAGE_PAGE_COMPRESSED and new field to indicate which compression algorithm is used. Support for LZ4, LZO, bzip2, LZMA and snappy has been added in addition to the existing LZ77 originally used by Row Compression. When a page is read, it is decompressed before it is on buffer pool.

IV. Multi-threaded Flush

MySQL storage engine InnoDB and XtraDB uses a caching region in memory named buffer pool. The buffer pool hosts

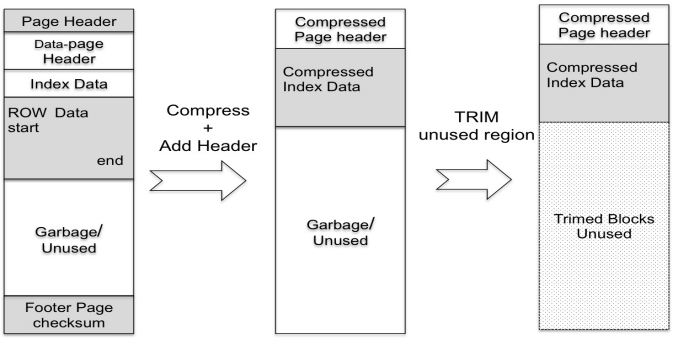


Fig. 2. NVM Compression on Sparse Address Space.

parts of the on-disk table space and system space for optimal performance. The buffer management is the key to service performance for query and update requests. Buffer pool comprises of data-pages clean and dirty, which have to be committed to the underlying media atomically. Buffer pool pages are flushed asynchronously using POSIX interfaces (AIO) to the underlying system, allowing for various methods of pool management including LRU and percentile dirty in cache for write intensive workload. In addition to delayed updates to underlying media to allow for multiple overwrites and updates to the table space.

NVM page compression is designed to enable compression only at point of Flush of dirty pages. In order to perform efficient compression, Multiple Threaded Flush (MT-Flush) framework was designed. The framework takes into consideration the system setup core count for any compression workloads, based on the available system resources the buffer-pool is split for optimal flush operation. The framework essentially allows for optimal use of available CPU compute cycles to perform compression using the specified compression algorithm on individual pages. Threads created as part of the framework allow for sets of data pages being flushed to be compressed in parallel thus reducing compression related latency, while maintaining optimal I/O queue depth critical for optimal storage system performance.

This new feature is implemented as traditional producer multiple consumers concept like:

- Work tasks are inserted into the work-queue (wq)
- The completion thread will look at the operation type and in case of a WRITE do the compression
- Operation completions get posted to return queue wr_cq.

Actual producer is single threaded and pseudocode is presented in Algorithm 1.

Furthermore, there is a configurable number of consumers (innodb_mtflush_threads) doing both compression and actual IO requests and their pseudocode is presented in Algorithm 2.

V. Evaluation

In this section we evaluate NVM Compression across various metrics ranging from populate time to performance. All experiments were conducted on Intel Xeon E5-2690 @ 2.9GHz CPU containing 2 sockets with 8 cores each using hyper threading, thus 32 total cores and Linux 3.4.12 with 132G main memory. The database is stored on a Fusion-io ioDrive3. The database filesystem is using NVMFS and all test logs and outputs are stored on second ioDrive using EXT4.

Algorithm 1 Producer

```
1:  while not shutdown do
2:      sleep so that one iteration takes roughly one second
3:      if flush LRU then
4:          for each buffer pool instance send a work item to flush LRU scan depth
            pages in chunks do
5:              send work items to multi-threaded flush work threads
6:              wait until we have received reply for all work items
7:          end for
8:      else if flush flush list then
9:          calculate target number of pages to flush
10:         for each instance set up a work item to flush (target number / # of instances)
            number pages do
11:             send work items to multi-threaded flush work thread
12:             wait until we have received reply for all items
13:         end for
14:     end if
15: end while
```

Algorithm 2 Consumers

```
1:  while not shutdown do
2:      wait until a work item is received from work queue
3:      if work_item type is EXIT then
4:          insert a reply message to return queue
5:          pthread_exit();
6:      else if work_item type is WRITE then
7:          call buf_mtflu_flush_pool_instance() for this work_item
8:          when we reach to os layer (os0file.cc) we compress the page if
9:          table uses page compression (fil0pagecompress.cc)
10:         set up reply message containing number of flushed pages
11:         insert a reply message to return queue
12:     end if
13: end while
```

We have selected following benchmarks:

- LinkBench [2] [2] which is based on traces from production databases that store social graph data at Facebook, a major social network. LinkBench provides a realistic and challenging test for persistent storage of social and web service data. The LinkBench measure phase uses 64 client threads and we use maxtime 86300 seconds, i.e. 24 hours, and warm-up is 180 seconds.
- Percona provides [3] an on-line transaction processing (OLTP) benchmark which is an implementation of TPC Benchmark C [19] like workload. Approved in July of 1992, TPC Benchmark C is an on-line transaction processing (OLTP) benchmark. TPC-C is more complex than previous OLTP benchmarks such as TPC-A because of its multiple transaction types, more complex database and overall execution structure. TPC-C involves a mix of five concurrent transactions of different types and complexity either executed on-line or queued for deferred execution. The database is comprised of nine types of tables with a wide range of record and population sizes. TPC-C is measured in transactions per minute (TpmC). For

[2]Source code can be obtained by git clone https://github.com/facebook/linkbench.git

[3]Source code can be obtained by bzr branch lp: percona-dev/perconatools/tpcc-mysql

TPC-C We use the default number of threads, i.e. 64 client threads (if not something else mentioned) and we have used time limit 10800 seconds, i.e. 3 hours, and warmup is 30 seconds.

We are using LinkBench with 10x database size i.e. about 100G and TPC-C with 1000 warehouses. InnoDB buffer pool is set to 50G, thus on both benchmarks the database does not fit in main-memory. We are using MariaDB 10.0.12 [4].

Only difference on configuration is that row-compressed (i.e. ROW_FORMAT=COMPRESSED) and uncompressed tables do not use atomic writes, trim and multi-threaded flush and they use doublewrite buffer. We will use InnoDB storage engine (5.6.15) on following tests.

Fusion-io ioDrive 3 is about 25 times faster compared to modern hard disk. This is so significant difference we decided that on rest of the results only Fusion-IO ioDrive is used as storage.

A. Micro-Benchmarks

In this paper we have presented atomic writes, page compression and persistent trims methods. In Figure 3 we present performance differences when different parts of these methods are enabled. We used LinkBench with 10x database and 10h measure time on all tests. Firstly, uncompressed tables not using atomic writes are compared to uncompressed tables using atomic writes. Atomic writes increases performance about 13%. Similarly, we experimented row compressed tables not using atomic writes and using atomic writes, however performance increase is only about 4% and could result on statistical fluctuations. For page compressed tables atomic writes increase the performance about 3%. Persistent trims are used only on page compressed tables and using persistent trims increase the performance about 7%.

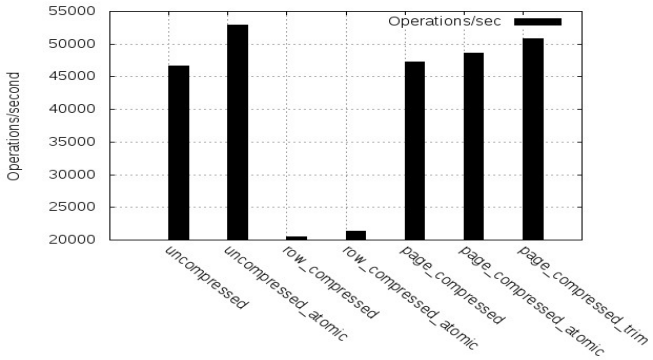


Fig. 3. Atomic writes, page compression and persistent trim compared.

B. LinkBench Evaluation

We start discussion on experimental results by showing time to populate LinkBench 10x database (100G) in Figure 4.

Clearly loading a row compressed tables take significantly longer than uncompressed or page compressed tables. We did not include loading time of page compressed tables using compression method lzma, because it loading time was 7800 seconds this could be from fact that database pages are relative small 16KB. Loading page compressed tables is faster than uncompressed at least when lz4, lzo and bzip2 is used. Clearly, bzip2 compression method provided the fastest compression speed. Compression speed is not the only significant factor when choosing compression method. Therefore, in Figure 5 we compare storage saving using different methods.

[4]Source code can be obtained by bzr branch lp:maria/10.0-FusionIO

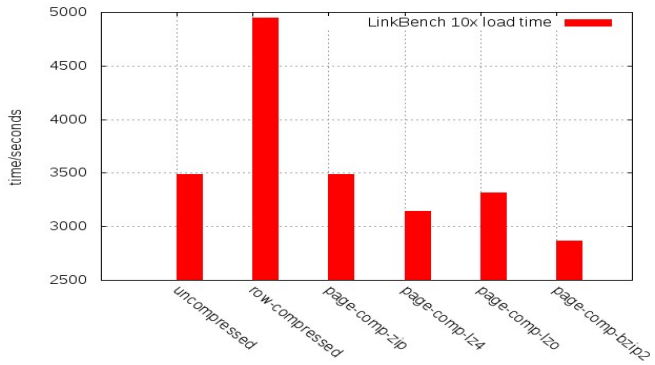


Fig. 4. Load times.

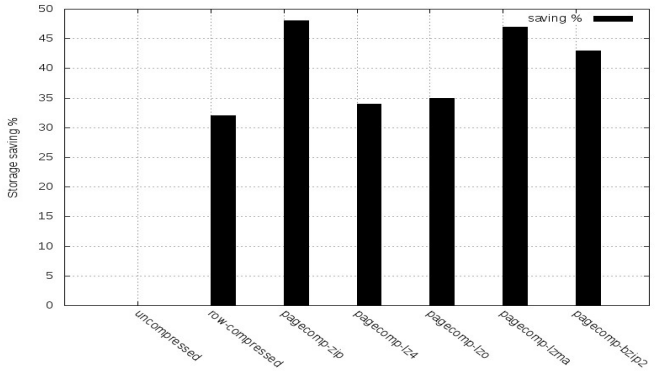


Fig. 5. Database size after load and measure.

Row-compressed and page compressed using lz4 or lzo offer similar compression savings compared to uncompressed. There is additional storage saving when using zip, lzma or bzip2 compression algorithms and lzma proved to be a little bit better. Compression can save between 32–48% of storage space when compared to uncompressed tables. Furthermore, page compression can save another 3-23% of storage space depending on used compression method. However, compression efficiency is not also the only deciding factor, since compression and decompression speed is also significant factor when choosing compression algorithm. In Figure 6 is shown number of LinkBench operations performed per second over the 24h measure time.

Now there is significant differences between compression and decompression speeds on different compression methods. Clearly, lz4 provides best compression and decompression speed matching almost the speed of uncompressed workload results. Difference between uncompressed and lz4 compressed is 6% meaning about 1800 operations in second, thus inside

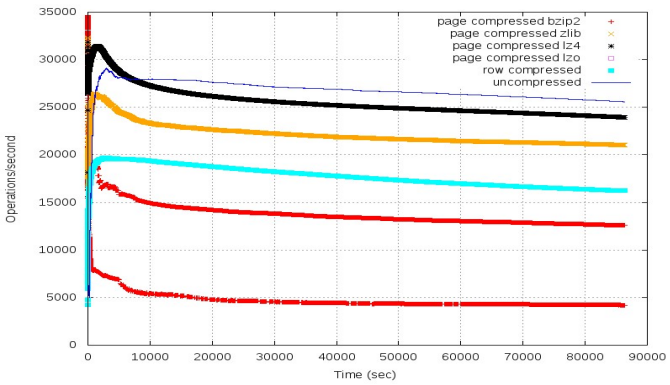


Fig. 6. LinkBench operations/second on measure phase.

a statistical variation. However, performance difference between page compressed tables using lz4 compared with row-compressed tables is significant 30% thus about 8000 operations in second. Clearly bzip2 and lzma could not offer very fast compression and decompression speed for this benchmark.

The average latencies of NVM Compression are generally comparable to those of Uncompressed (see Figure 7). Row Compression throughput is much lower compared to NVM Compression and Uncompressed. The 99th percentile latencies for NVM Compression are 2.5x to 5.5x lower than that for the default Row Compression. The NVM Compression latencies are also close to that of the Uncompressed workload, with lower latencies than the Uncompressed workload for some operations (Update Node, Deleted Node and Get Node).

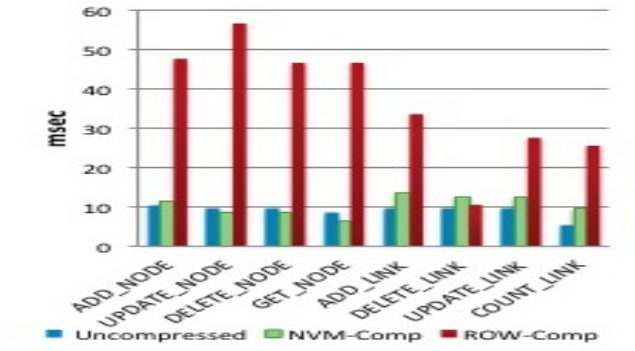


Fig. 7. Mean operation latency (mSec).

The above results confirm that the hybrid NVM Compression approach can deliver benefits by removing some of the complexities inherent in the Row Compression scheme. However, NVM Compression is also able to keep up with, and occasionally outperform, the Uncompressed configuration. To better understand this, we measured the write behavior of all three configurations. Figure 8 highlights the data written per linkbench operation measured across 30 second intervals during a six hour run. NVM Compression writes far less data than uncompressed, which leads to better performance in the fast device over time since less garbage collection has to occur. We also found, that although Row- Compression stores less total data than Uncompressed, it also generates more data writes per unit of operation than Uncompressed or NVM Compression, which likely is a result of insert failures leading to node splits and allocation map re-balance 2 requiring additional writes thereby further contributes to its poor performance.

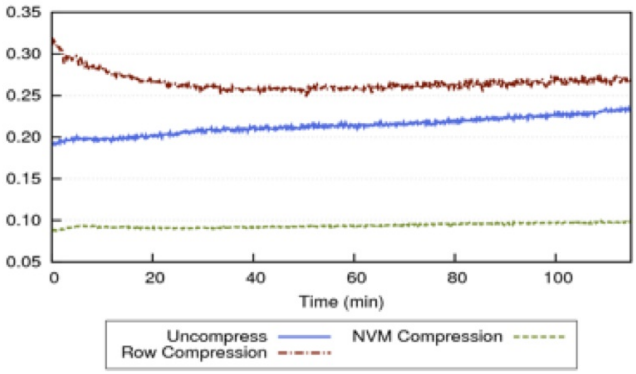


Fig. 8. Cost of Write KB/OPs.

C. OLTP-like Evaluation

Next we study compression performance for an On-Line Transaction Processing (OLTP) benchmark. This TPC-C like workload involves a mix of five concurrent transaction types executed on-line or queued for deferred execution. The database is comprised of nine tables with a wide range of record and population sizes. Results are measured in terms of transactions per minute (TpmC). In Figure 9 we present TpmC results when number of client threads is 32.

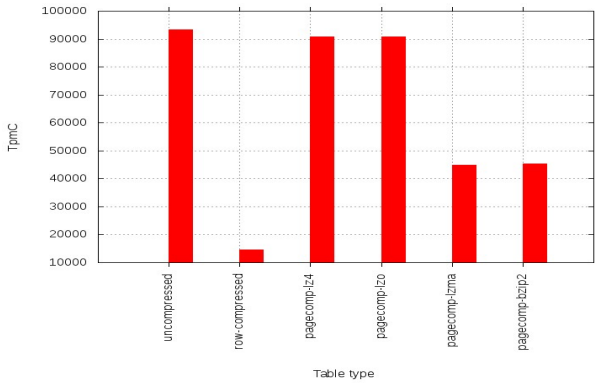


Fig. 9. Percona TPC-C TpmC results.

Clearly row-compressed provides significantly lower performance while lz4 and lzo could provide similar performance compared with uncompressed workload. Difference between uncompressed and lz4 page compressed is about 2500 TpmC i.e. less than 3%. Page compression using lz4 provides about 6x performance compared with row-compressed. Again lzo and bzip2 methods provided significantly lower performance but still better performance compared to row-compressed.

Figure 10 presents the number of New Order transactions in second when number of TPC-C clients threads is 32.

Lz4 and lzo methods provide similar performance compared to uncompressed workload and row-compressed is significantly slower than any of page-compressed methods.

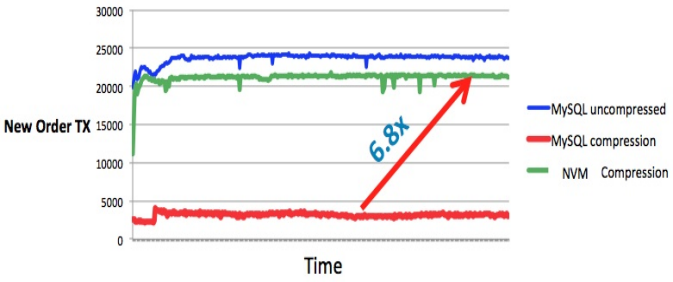


Fig. 10. TPC-C New Order transactions/second results.

Difference between lz4 and lzo methods compared to uncompressed is between 0–6% i.e between 50-1000 transactions/second. Both lz4 or lzo can provide between 5-7x performance compared to row-compressed. Finally, Figure 11 shows TpmC results when number of the TPC-C client threads are varied from 4 to 512.

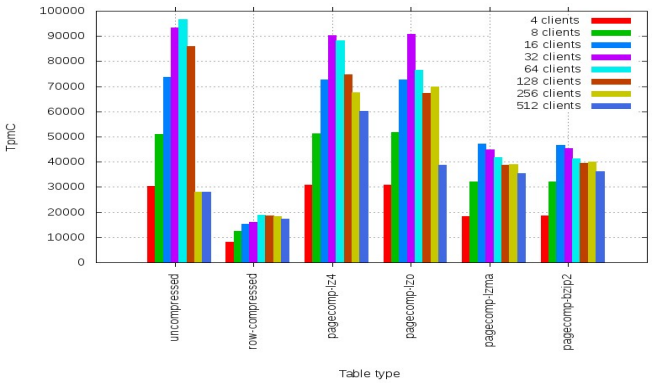


Fig. 11. Percona TPC-C TpmC when number of client threads are varied.

Again lz4 and lzo methods provide similar performance compared to uncompressed workload and even better performance when number of client threads is higher than number of the cores in system (32). Performance of lz4 or lzo methods is between 3-5x better compared to row-compressed. Similarly, lzma and bzip2 offer lower performance compared to other page-compression methods but they can also provide better performance than uncompressed when number of client threads is higher than 128.

VI. Conclusions

NVM-Compression is designed to combine the best of application level compression and flash aware integration. The block management and high-speed garbage collection of flash are leveraged through FTL primitives. File system support ensures that standard database deployment practices, such as placing tables in files, are preserved. The performance results are dramatic, with performance close to or sometimes exceeding uncompressed on both Linkbench and OLTP workloads due to less garbage collection.

NVM Compression used in combination with Atomic Writes also improves endurance by about 4x. NVM compression has been released by all three MySQL distributions, MariaDB™ [5] Percona™ [6] and Oracle™ [7]. MariaDB has been the reference implementation and all development/analysis has been done on that code base. Other MySQL vendors have followed suit and added their own implementation based on the work by MariaDB. The code for the MySQL storage engine implementations of NVM Compression is available in open source from all of the three distributions.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, pages 671–682, 2006.

[2] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: a database benchmark based on the Facebook social graph. In *SIGMOD Conference*, pages 1185–1196, 2013.

[3] Daniel Bartholomew. *Getting Started with MariaDB*. Packt Publishing, 2013.

[4] M. Callaghan. The effect of page size on InnoDB compression. https://www.facebook.com/note.php?note_id=10150348315455933.

[5] M Callaghan. Making InnoDB compression adaptive. https://www.facebook.com/note.php?note_id=10150345355665933.

[6] Peiquan Jin, Yi Ou, Theo Härder, and Zhi Li. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. *Data Knowl. Eng.*, 72:83–102, 2012.

[7] Jürgen Kaiser, Fabio Margaglia, and André Brinkmann. Extending SSD lifetime in database applications with page overwrites. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 11:1–11:12, 2013.

[8] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan S. Milojicic. Optimizing checkpoints using NVM as virtual memory. In *IPDPS*, pages 29–40, 2013.

[9] Sang-Won Lee, Bongki Moon, Chanik Park, Joo Young Hwang, and Kangnyeon Kim. Accelerating in-page logging with non-volatile memory. *IEEE Data Eng. Bull.*, 33(4):41–47, 2010.

[10] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM duet: unified working memory and persistent store architecture. In *ASPLOS*, pages 455–470, 2014.

[11] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, June 2014.

[12] Shuichi Oikawa. Towards new interface for non-volatile memory storage. In *PECCS*, pages 174–179, 2014.

[13] Shuichi Oikawa and Satoshi Miki. File-based memory management for non-volatile main memory. In *COMPSAC*, pages 559–568, 2013.

[14] Shuichi Oikawa and Satoshi Miki. Future non-volatile memory storage architecture and file system interface. In *CANDAR*, pages 389–392, 2013.

[15] N Ordulu. Getting InnoDB compression ready for Facebook scale. http://www.percona.com/live/mysql-conference-2012/sessions/getting-innodb-compression-ready-facebook-scale.

[16] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *HPCA*, pages 301–311, 2011.

[17] Youngjo Park and Jin-Soo Kim. zFTL: power-efficient data compression support for NAND flash-based consumer electronics devices *IEEE Trans. Consumer Electronics*, 57(3):1148–1156, 2011.

[18] Ilia Petrov, Robert Gottstein, Todor Ivanov, Daniel Bausch, and Alejandro P. Buchmann. Page size selection for OLTP databases on SSD storage. *JIDM*, 2(1):11–18, 2011.

[19] Francois Raab. TPC-C - the standard benchmark for online transaction processing (oltp). In *The Benchmark Handbook*. 1993.

[20] I. Rana. InnoDB compression improvements in MySQL 5.6. https://blogs.oracle.com/mysqlinnodb/entry/innodb_compression_improvements_in_mysql.

[21] SanDisk. NVMFS. http://itblog.sandisk.com/in-a-battle-of-hardware-software-innovation-comes-out-on-top.

[22] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[23] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. HEC: improving endurance of high performance flash-based cache devices. In *SYSTOR*, page 10, 2013.

[24] Keun Soo Yim, Hyokyung Bahn, and Kern Koh. A flash compression layer for smartmedia card systems. *IEEE Trans. Consumer Electronics*, 50(1):192–197, 2004.

[5] available at https://github.com/MariaDB/server/tree/10.0-FusionIO

[6] available at http://code.launchpad.net/~gl-az/percona-server/5.6-pagecomp_mtflush

[7] available at http://labs.mysql.com