Open Access

Open Journal of Databases (OJDB)
Volume 4, Issue 1, 2017
http://www.ronpub.com/ojdb
ISSN 2199-3459

An NVM Aware MariaDB Database System and Associated IO Workload on File Systems

Jan Lindström [A], Dhananjoy Das [B], Nick Piggin [C], Santhosh Konundinya [D], Torben Mathiasen [D], Nisha Talagala [B], Dulcardo Arteaga [B]

[A] MariaDB Corporation, Joensuu, Finland, jan.lindstrom@mariadb.com
[B] Parallel Machines, USA, firstname.lastname@ParallelMachines.com
[C] SanDisk (now IBM), npiggin@gmail.com
[D] Yellowbrick, firstname@yellowbrick.io

Abstract

MariaDB is a community-developed fork of the MySQL relational database management system and originally designed and implemented in order to use the traditional spinning disk architecture. With Non-Volatile memory (NVM) technology now in the forefront and main stream for server storage (Data centers), MariaDB addresses the need 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, improve 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. Various micro benchmark measurement and findings on sparse files call for required improvement in file systems for handling of punch hole operations on files.

Type of Paper and Keywords

Regular research paper: *MariaDB, Non-Volatile Memory, NVM, Compression, Flash Translation Layer, FTL, garbage collection, optimization, file system*

1 Introduction

Traditionally compression has been extensively used to save expensive resources of capacity and bandwidth. Compression is also extremely attractive for flash based systems to improve cost/capacity and to improve lifetime by reducing media writes. In non-volatile memory devices Flash Translation Layer (FTL) is an matching location to perform compression, because it already performs sophisticated data management [44, 32, 20]. A Flash Translation Layer is responsible for the mapping of Logical Block Address (LBA) updates into physical block updates (PBA), and thus can concurrently perform the compression. Flash devices have a quite high cost, lower capacity compared to traditional disk drives,

and lower write endurance [15, 42]. 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 transparent to applications [44]. Integrating compression to FTL may be ideal for flash devices, but benefits of application level compression originating from better knowledge of application patterns can be lost. Compression has been successfully used e.g. on key-value stores [1, 22]. Furthermore, non-volatile memory device drivers and file systems have started to appear [25, 27, 26].

Previous research on non-volatile memory storage have been concentrated on specific sub-systems of database architecture like checkpointing [16], combining main-memory and non-volatile memory [41, 27, 19], page size selection [33] or buffer replacement algorithm [13].

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:

- Design and implementation of a novel compression mechanism that combines the advantages of application level awareness and FTL integration.
- Integration of such an approach into an operating system stack through a combination of new interface primitives and file system support.
- Performance study showing that this approach can improve compression performance by 2-3x compared to the legacy Row based compression technique.
- Improving Transaction throughput of NVM Compression, which is within 5% of uncompressed workload performance.
- Improving overall storage efficiency by 19%, compared to the legacy Row based compression.
- Reducing overall transactional data writes to the underlying NVM media by 4x when used in combination of Atomic Writes, also reducing the transactional IO latency.
- A performance study of NVM Compression method on various file systems, calling for the needs and potential benefits of usage of NVM primitives by filesystems.

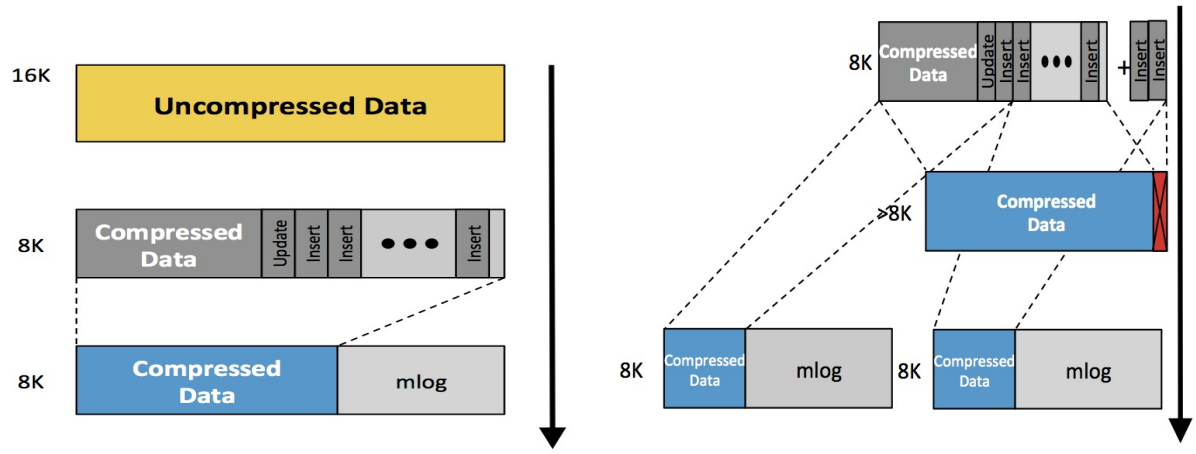
2 BACKGROUND

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] [4] 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), MySQL Row Compression compresses data rows into a predefined compressed data block. The size of the data block is defined by an administration 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 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. An adjacent page first 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 [6, 5, 28, 35].

[1] www.mariadb.org



(a) MySQL Row Compression.

(b) Insert Failure

Figure 1: Traditional MySQL Row Compression

3 NVM COMPRESSION

The goals of our compression solution, NVM Compression, are to deliver a 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 into 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.

Figure 2 shows the high level approach. NVM compression is applied to database pages and whole page content is compressed. As a result we have a new page where compressed page header is added. Rest of the page is garbage and unused. Finally, the trim operation is applied leaving a "hole", i.e. an empty location, in the remainder of the space allocated for the fully uncompressed block.

FTL responsibility on NVM compression is to perform thin provisioning, i.e. allocating physical capacity only to the populated virtual addresses. The virtual address holes are unpopulated. Under this model, the FTL will have populated virtual addresses and some empty virtual addresses. FTL garbage collection will naturally coalesce the populated addresses in physical space and thus allow for re-provisioning of the available physical space to be used for new writes.

As database records are overwritten, the size of compressed data content can change. In particular, blocks may compress to smaller sizes than they did previously. Since, in the NVM Compression design, re-compressed blocks are always rewritten to the same virtual locations, it becomes necessary to punch a hole in the address space, i.e. creating a hole in the virtual address space where previously there may have been data.

To benefit from the FTL thin provisioning as discussed above, we need a way to expose that capability to the layer above, in this case, a user space application. We expose native characteristics of flash translation layers through a series of primitives that are exported by the FTL. This is similar in approach to other flash specific optimizations for KV stores, File Systems and Caches as described in [14, 22, 42, 36]:

- We modify the MySQL database to utilize these flash primitives in order to create a new compression design.
- In order to allow the database to operate on regular files (as is the norm for nearly all MySQL deployments), we modify a file system to effectively export the primitives from the flash translation layer to the user space application.
- To ease the integration process, we also map these primitives to existing Linux system calls.

Specifically, NVM Compression uses three FTL exposed primitives: a sparse address space, persistent TRIMs and atomic writes. These flash characteristics are then exported by the Non-Volatile Memory File System

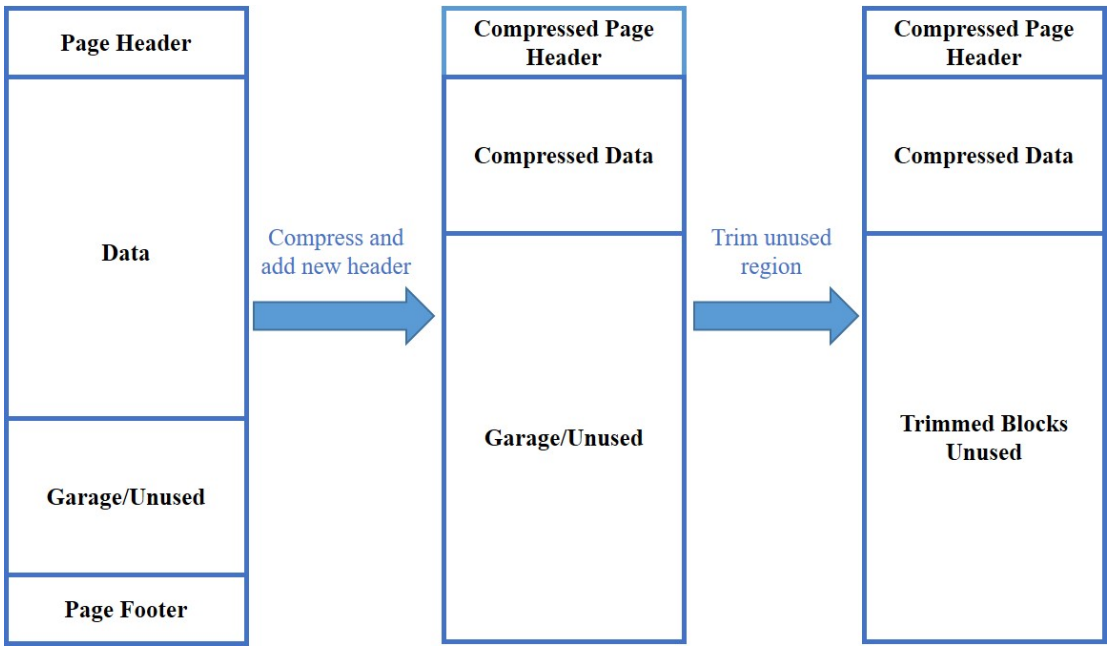


Figure 2: NVM Compression on sparse address space

Table 1: NVM Primitives

Primitives	Detail
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.

(NVMFS) [14, 39] to a POSIX compliant file system that is developed specifically to efficiently access flash and export native flash control to user space applications. NVMFS is an extension of the work done in [14]. MySQL is then modified to support a new compression mode, NVM compression, which uses the primitives on files stored in the NVM file system.

The NVMFS POSIX compliant file system [8] re-exports the functionality as file level interfaces as described in Table Table 1. The sparseness required for NVM Compression is provided through sparse files in NVMFS. The files are pre-allocated to be the size required for uncompressed data. As data is compressed during database operation, holes are created within the file through the *fallocate(PUNCH HOLE)* Linux system call operation.

It is important to note that the NVM Compression design does not specifically require NVMFS [14, 39] and also work on other file systems such as EXT4 [7] and XFS [40]. However, NVMFS is able to process operations like Persistent TRIM very efficiently since its design is also based on the FTL sparse address space (see Figure 2). NVMFS offloads the complexity of remapping and translation to the FTL, leveraging the capabilities of the underlying flash device. This architecture (see Figure 3) implies that NVMFS does not need any additional data operations to optimize file system layouts for sparse files.

NVMFS exports the Persistent TRIM capability through the *fallocate(PUNCH HOLE)* operation. This operation is converted by NVMFS into a PTRIM primitive to the underlying FTL. Atomic Writes are exported as conventional writes, which are configured to be atomic by default when issued to specific files. 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 [31]. 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';
```

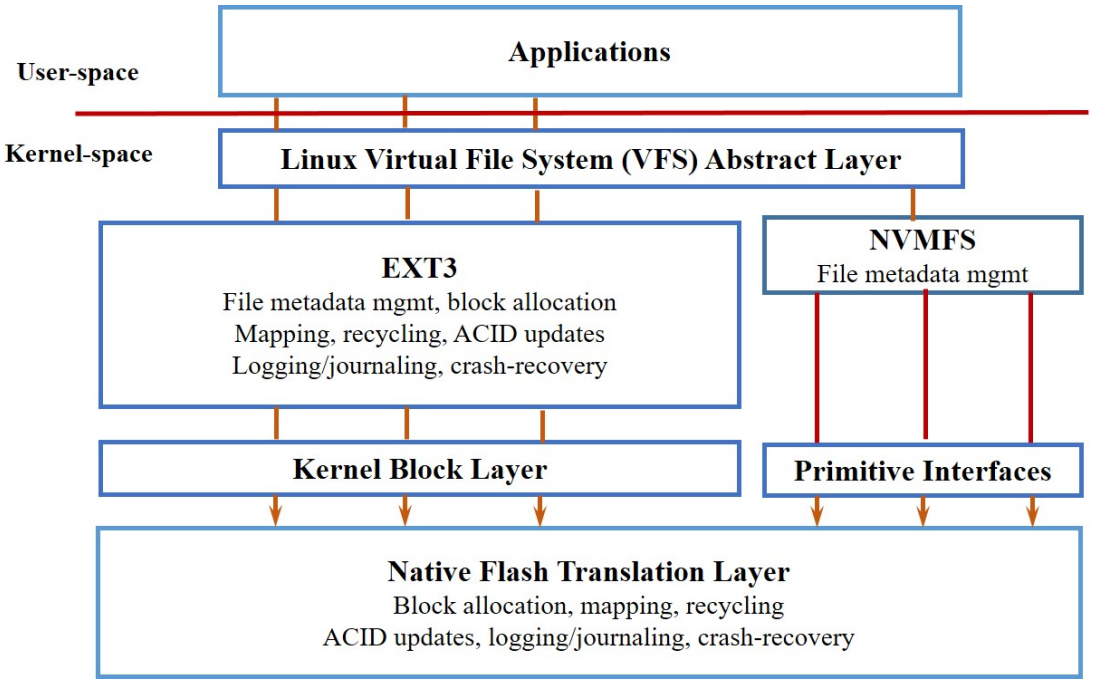


Figure 3: NVMFS architecture

Instead of storing both compressed and uncompressed pages in the buffer pool, we store only a uncompressed 16KB pages in the buffer pool. This avoids very complex logic when a page needs to be re-compressed or when adding 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 also provides a way to define tables that should use page compression and what compression level is used if available by used compression algorithm:

```
1: create table atomictable (x int unsigned not null
   primary key)
2: engine = innodb
3: page_compressed                                  =
   [none—lz4—lzo—snappy—zlib]
   page_compression_level = 6
4: data directory = '/mnt/fusionio/data';
```

All these are stored into InnoDB data dictionary persistently and all values can be changed with the normal alter table SQL-clause. If a compression level or algorithm is not specified, a system global value is used. Compression level can be specified using the innodb_compression_level option and algorithm with innodb_compression_method configuration variables. Both of these can be changed dynamically without a 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 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. A PTRIM operation is issued for the remainder of the address range (using *fallocate(PUNCH HOLE)*) to inform NVMFS and the FTL of that the remainder of the virtual addresses are now empty and can be garbage collected as appropriate.

This has been implemented with the special page type FIL_PAGE_PAGE_COMPRESSED and a new field to indicate which compression algorithm is used. Support for LZ4[2], LZO[3], bzip2[4], LZMA [5] and snappy[6] has been added in addition to the existing LZ77[45] originally used by Row Compression. When a page is read, it is decompressed before it is stored in the buffer pool.

4 Multi-Threaded Flush

MySQL storage engines InnoDB and XtraDB use a caching region in a memory named buffer pool. The buffer pool hosts parts of the on-disk table space and system space for optimal performance. The buffer management is the key to service performance for query

[2] http://lz4.github.io/lz4/
[3] http://www.oberhumer.com/opensource/lzo/
[4] http://www.bzip.org/
[5] http://tukaani.org/xz/
[6] http://google.github.io/snappy/

and update requests. The 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, underlying media allows for multiple overwrites and updates to the table space.

NVM page compression is designed to enable compression only at the point of the 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, and 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 an optimal I/O queue depth, which is critical for an optimal storage system performance.

This new feature is implemented as a 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.

The 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.

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 we compress the page if
9:      table uses page compression
10:     set up reply message containing number of flushed pages
11:     insert a reply message to return queue
12:   end if
13: end while
```

5 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 NVMFS and all test logs and outputs are stored on the second ioDrive using EXT4.

We have selected following benchmarks:

LinkBench[7] [3]: which is based on the traces of production databases that store social graph data from 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 the measure time is 86300 seconds (24 hours), and the warm-up time is 180 seconds.

Percona[8] provides an on-line transaction processing (OLTP) benchmark, which is an implementation of TPC Benchmark C [34]. Approved in July of 1992, TPC Benchmark C (TPC-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 TPC-C, we use the default number of threads, i.e. 64 client threads (if not something else mentioned) and the measure time is 10800 seconds (3 hours), and the warm-up time is 30 seconds.

We use 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. The database management system, which we use, is MariaDB 10.0.12[9]. 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.

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

5.1 Micro-Benchmarks

In this paper we have presented atomic writes, page compression and persistent trims methods. In Figure 4 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.

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

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

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

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%.

5.2 LinkBench Evaluation

We start discussion on experimental results by showing time to populate LinkBench 10x database (100G) in Figure 5. 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 its loading time was 7800 seconds this could be from the fact that database pages are relative small 16KB. Loading page compressed tables is faster than uncompressed at least when lz4, lzo and bzip2 are used. Clearly, bzip2 compression method provided the fastest compression speed. Compression speed is not the only significant factor when choosing compression methods. Therefore, in Figure 6 we compare storage saving using different methods.

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 also not the only deciding factor, since compression and decompression speed is also a significant factor when choosing compression algorithms. In Figure 7 is shown the 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. The difference between uncompressed and lz4 compressed is 6% meaning about 1800 operations in second, thus inside a statistical variation. However, performance difference between page compressed tables using lz4 and row-compressed tables is 30% meaning about 8000 operations in second. Clearly bzip2 and lzma could not

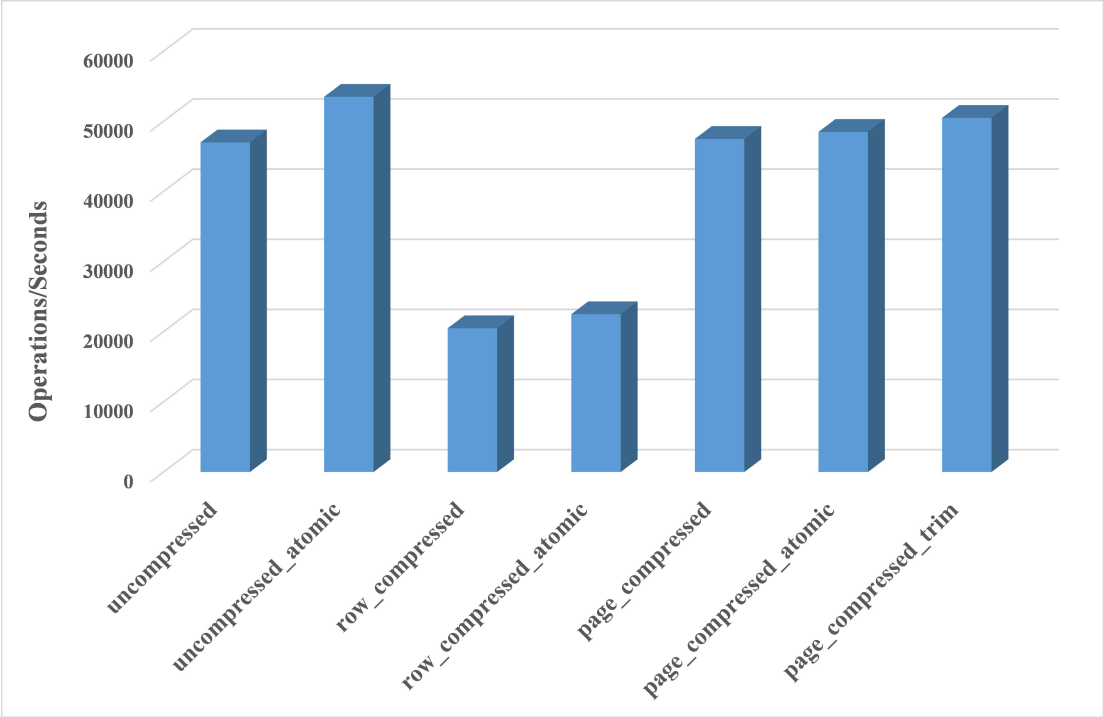


Figure 4: Atomic writes, page compression and persistent trim compared

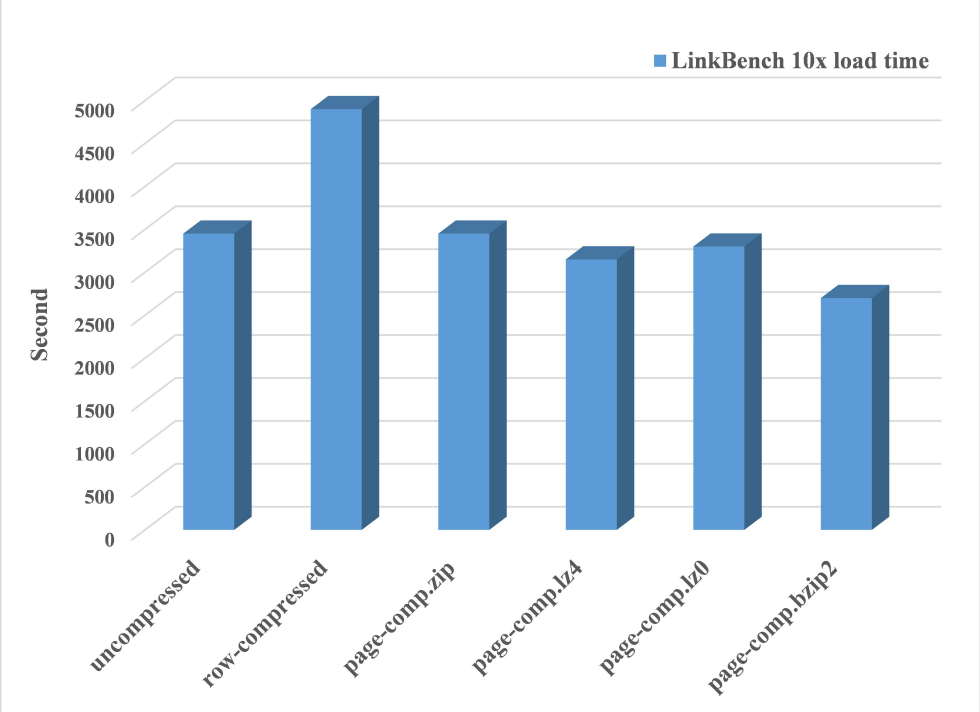


Figure 5: Load time of the LinkBench 100G database

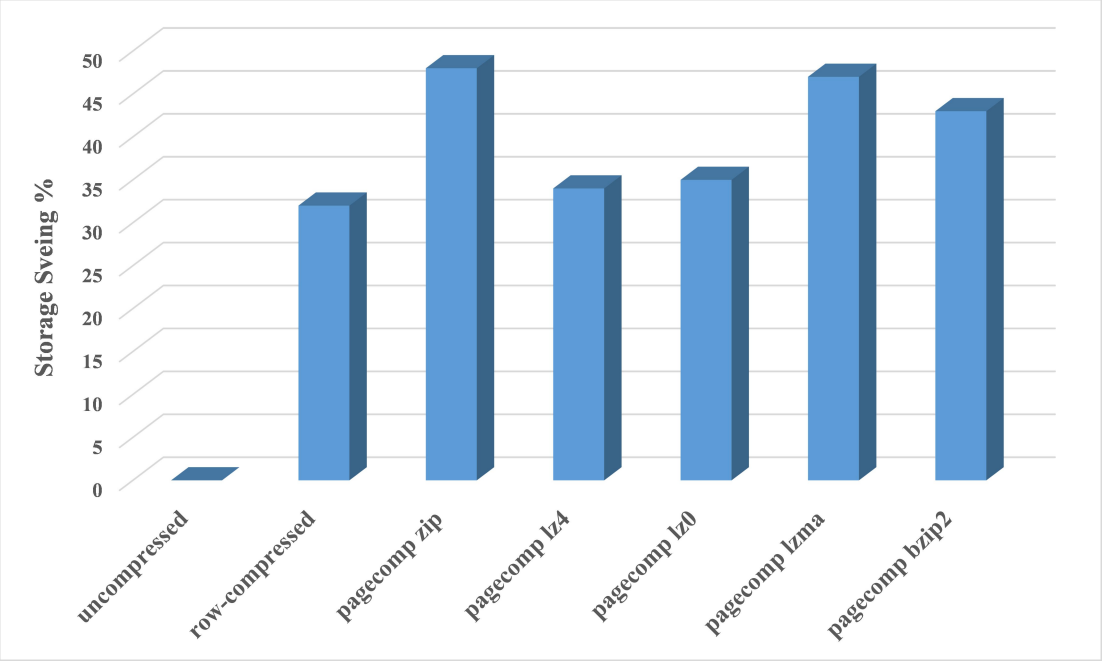


Figure 6: Storage saving of different methods

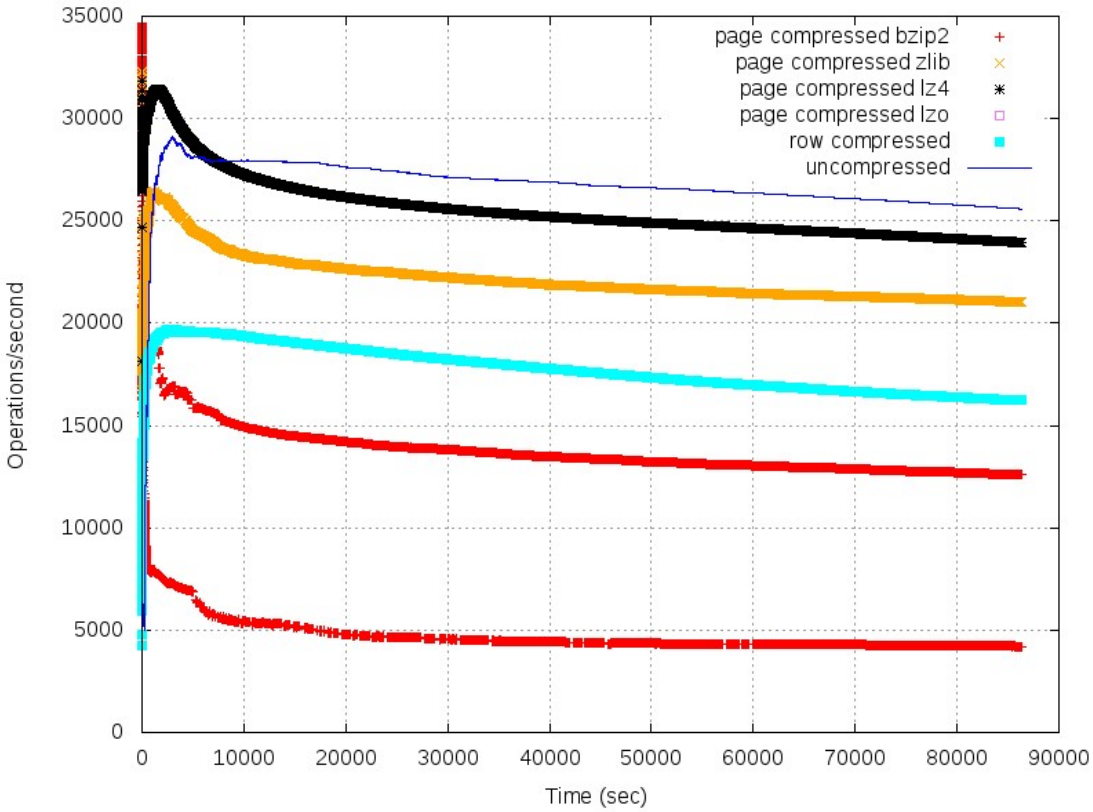


Figure 7: Number of LinkBench operations per second at the measure phase

offer very fast compression and decompression speed for the LinkBench benchmark.

The average latencies of NVM Compression are generally comparable to those of Uncompressed (see Figure 8). Row Compression throughput is much lower compared to NVM Compression and Uncompressed. The 99th percentile latency for NVM Compression are 2.5x to 5.5x lower than that for the default Row Compression. The NVM Compression latency is also close to that of the Uncompressed workload, with lower latency than the Uncompressed workload for some operations like Update Node, Deleted Node and Get Node.

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 9 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 data than Uncompressed, it also generates more data writes per unit of operation than Uncompressed or NVM Compression. This is likely a result of insert failures leading to node splits and allocation maps requiring additional writes, and thereby further contributes to its poor performance.

5.3 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 10 we present TpmC results when number of client threads is 32.

Clearly, the row-compressed method provides a significantly lower performance while lz4 and lzo could provide a 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 a significantly lower performance but is still better compared to Row-Compressed.

Figure 11 presents the number of New Order transactions per second when the number of TPC-C clients threads is 32. Lz4 and lzo methods provide a similar performance compared to uncompressed workload, and Row-Compressed is significantly slower than any of page-ompressed methods. 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 12 shows TpmC results when the number of the TPC-C client threads are varied from 4 to 512. Again lz4 and lzo methods provide a similar performance compared to uncompressed workloads, and even a better performance when the number of client threads is higher than the number of the cores in the system.

The performance of lz4 or lzo methods is between 3-5x better compared to Row-Compressed. Similarly, lzma and bzip2 offer a lower performance compared to other page-compression methods, but they can also provide better performance than uncompressed when the number of client threads is higher than 128.

5.4 Page Size Evaluation

MariaDB's InnoDB and XtraDB storage engines use 16Kb page size by default. However, MariaDB offers a support of up to 64Kb page sizes for uncompressed and page compressed tables in MariaDB 10.1. In the following we present the measurement with MariaDB 10.1.9 using LinkBench with 20x database (about 200Gb) and 5 hours of measure phase. The buffer pool size used is 20Gb. These tests are run with different hardware from previously, i.e. CentOS Linux release 7.1.1503 (Core) using the 3.10.0-229.el7.x86_64 Linux kernel, ioMemory SX300-1600 with VSL driver 4.2.1 build 1137 and NVMFS 1.1.1. Thus, these results can not directly be compared to the earlier results in this paper.

In Figure 13 we compare total database storage sizes using uncompressed, lz4 compressed, lzo compressed and zlib compressed tables. Other supported compression methods lzma, bzip2 and snappy were not suitable for this kind of workload. We measured total database storage sizes after the LinkBench load and 5 hour measure phase. Clearly, zlib provides the best compression and lz4, and lzo provide a similar compression efficiency.

In Figure 14 we compare the results from LinkBench 5 hour measure phase comparing the reported operations per second at the end of measure phase. These results show that lz4 provides the best performance in

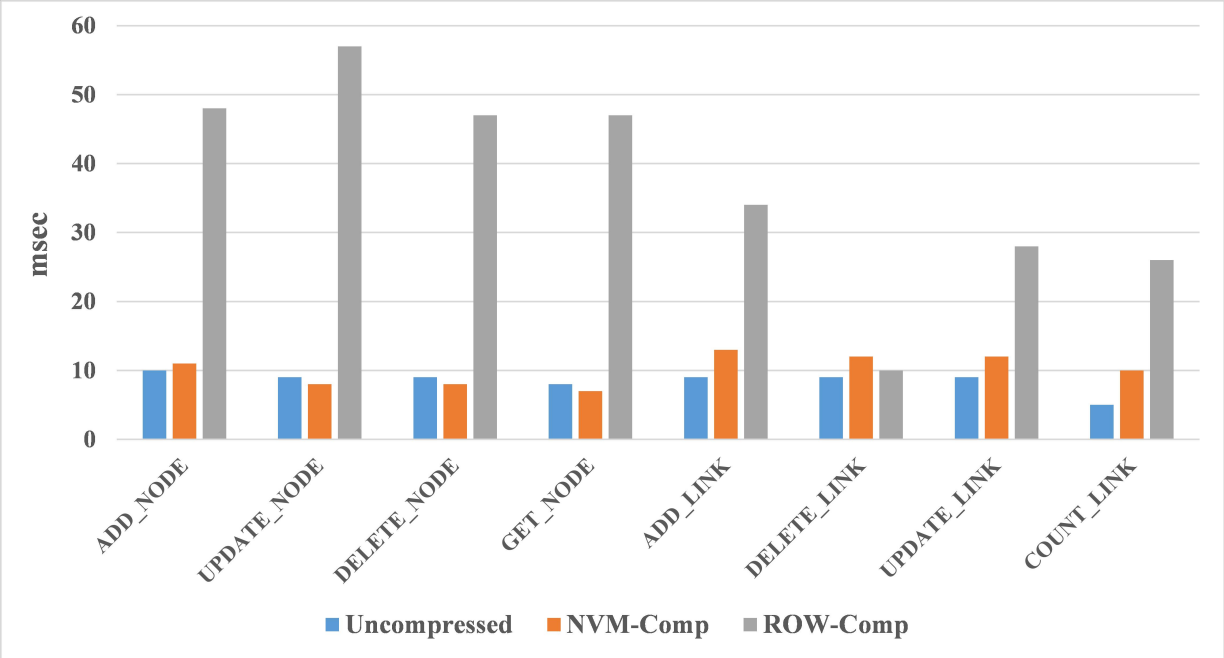


Figure 8: The average latencies of operations

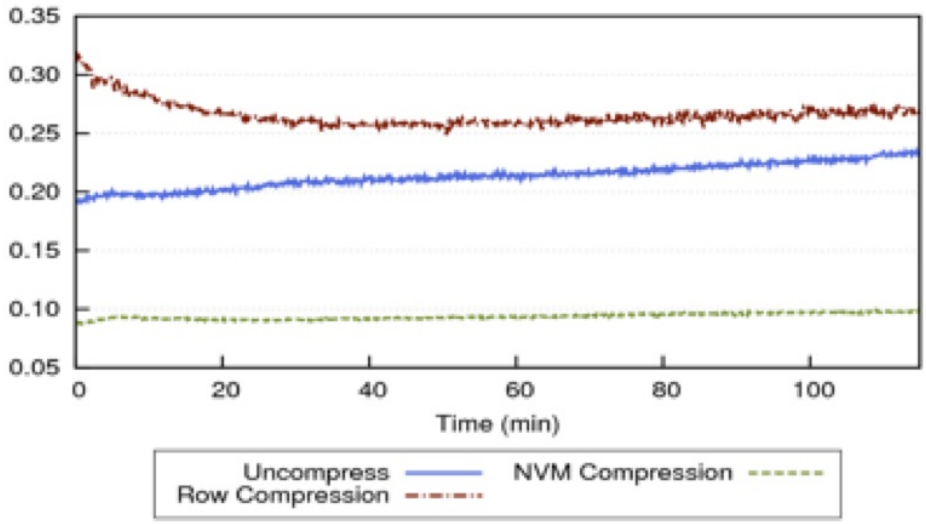


Figure 9: Data amount (Kilobytes) written per linkbench operation (KB/OP) measured across 30 second intervals during a six hour run

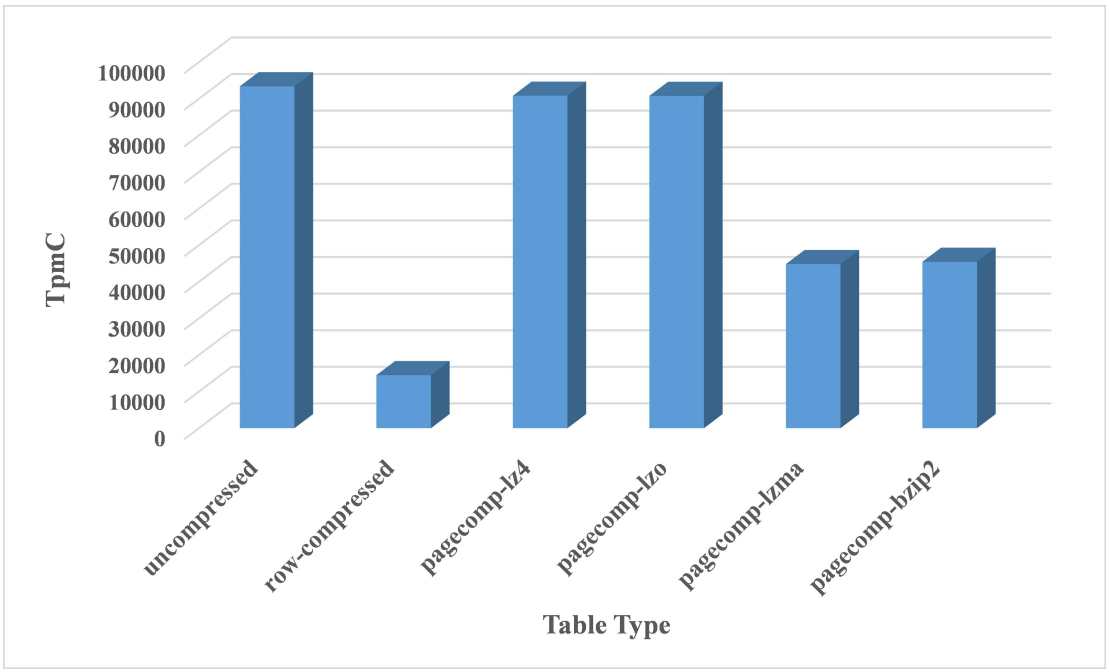


Figure 10: TPC-C compression performance of Percona measured in transactions per minute (TpmC) with 32 threads

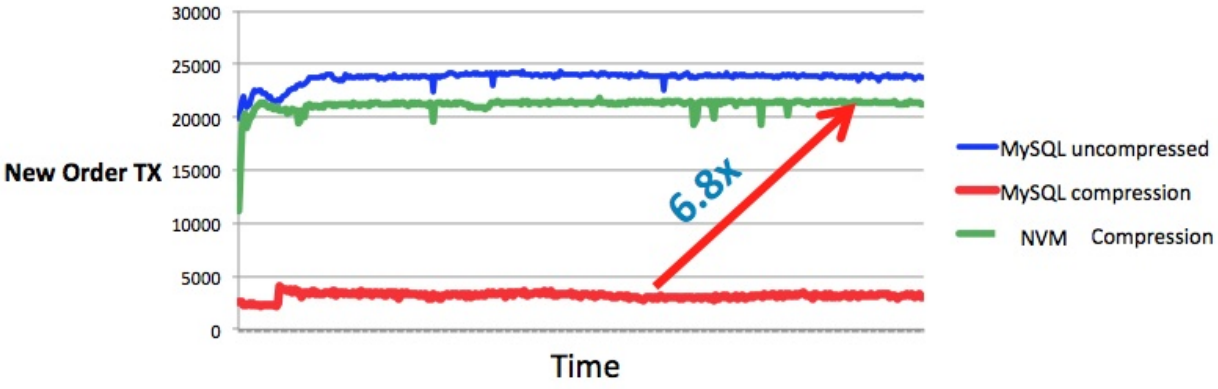


Figure 11: The number of New Order transactions of TPC-C per second

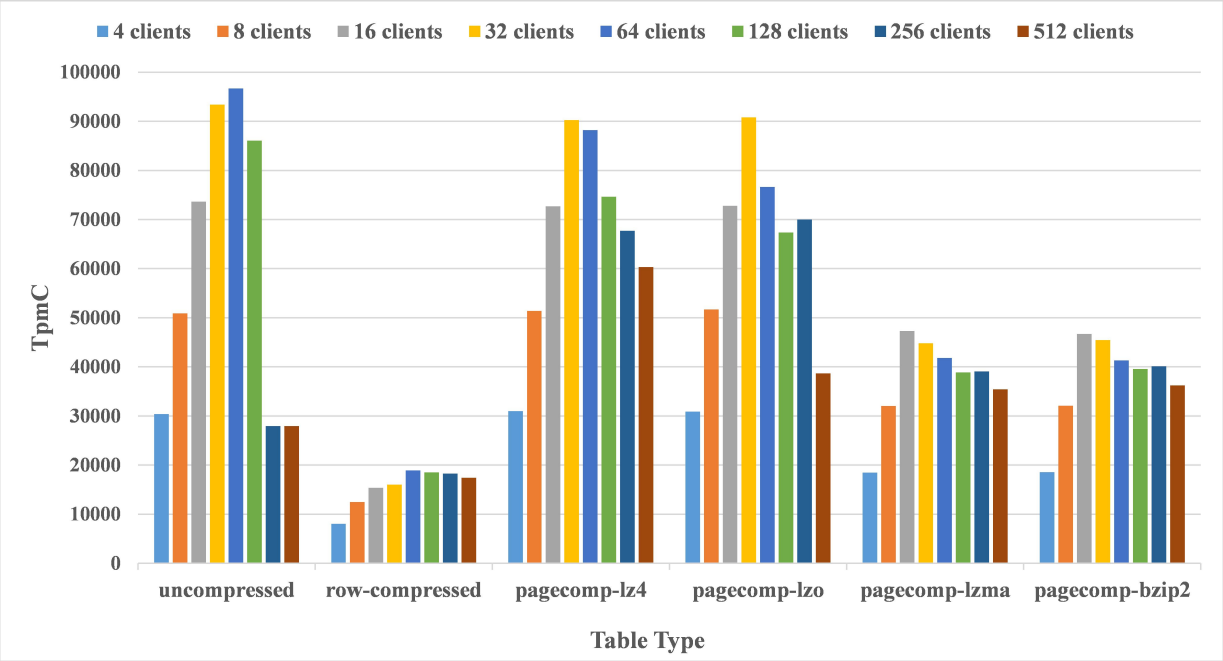


Figure 12: TPC-C compression performance of Percona measured in transactions per minute (TpmC) with different numbers of theards

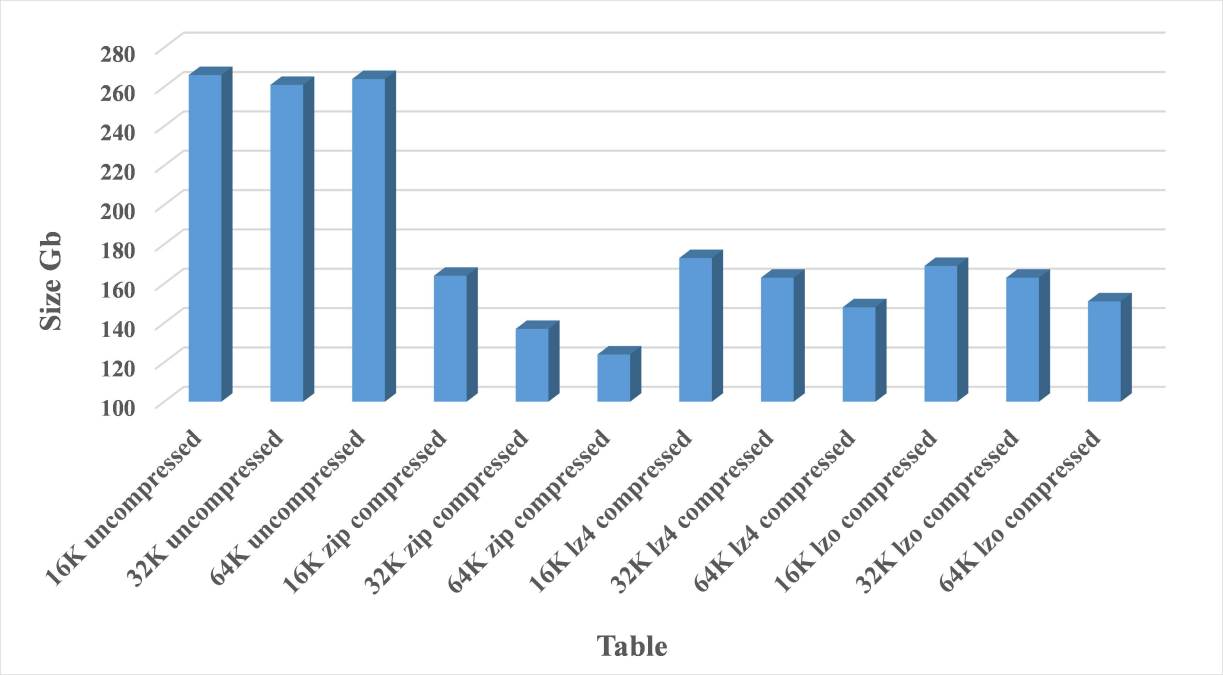


Figure 13: Database sizes after LinkBench database loading and 5-hour measure phase

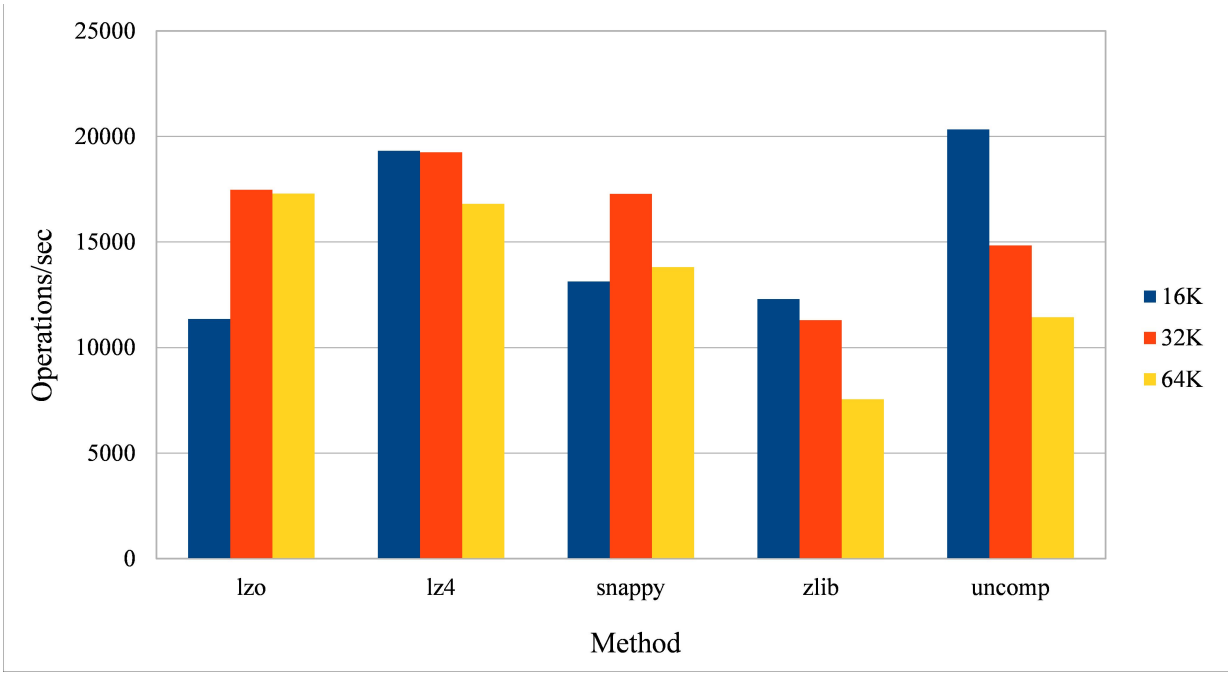


Figure 14: The average number of LinkBench operations per second after a 5-hour measure phase

different page sizes. The difference between lz4 and uncompressed is only 5% at 16Kb and lz4, and actually outperforms uncompressed when the page size is larger than 16Kb.

In Figure 15 we show the results of the operations per second over time using LinkBench 5 hour measure phase comparing lz4 compressed tables and uncompressed tables. These results clearly show that after the buffer pool is full both lz4 compressed tables and uncompressed tables provide an almost identical performance (the difference is about 5%). When the database page size is increased from the default 16Kb to 32Kb or 64Kb, lz4 compressed tables clearly outperform uncompressed tables. This is mainly due to fact that there is significantly less data written to the media.

Finally in Figure 16 we show the mean latencies of different LinkBench benchmark operations using lz4 compressed tables and uncompressed tables with different database page sizes.

5.5 File system comparison

In order to understand and evaluate the value proposition of an IO model exporting sparse and other primitives to the application, we performed some micro-benchmark experiments. The evaluation was restricted to NVMFS, XFS and EXT4 file systems, and the underlying block device used was the identical flash device in all the experiments.

Figure 17 presents the result of TRIM operation. The micro benchmark simulates the I/O patterns of NVM Compression on the underlying file systems for persisting the data set. This micro benchmark workload comprises of writes of size 16KB (default page size), and the TRIM operation of 4KB. The micro-benchmark evaluates the performance of the various file systems with a workload like MySQL NVM compression. In this test EXT4 provides the best latency for the TRIM operation, NVMFS is about 2x slower than EXT4. However, NVMFS is 4x faster compared to XFS.

Figure 18 presents the og write operation. In this micro benchmark, we measure the write amplification of the underlying file systems using the log write operation of 4KB followed by the fdatasync operation. Micro benchmark on NVMFS has a number of writes comparable to the operations on a raw block device (write operations without a file system), XFS has 2x write amplification compared to NVMFS and EXT4 has 5x write amplification. These write amplifications can contribute to lower transaction throughput: (i) due to higher transaction latency for individual transaction commits, (ii) as the system ages due to significant resources being used for operations like garbage collection and inhibiting IO throughput on the underlying block device, thu further impacting the limited number of write cycles on the underlying flash storage.

In Figure 19 we compare NVM compression on

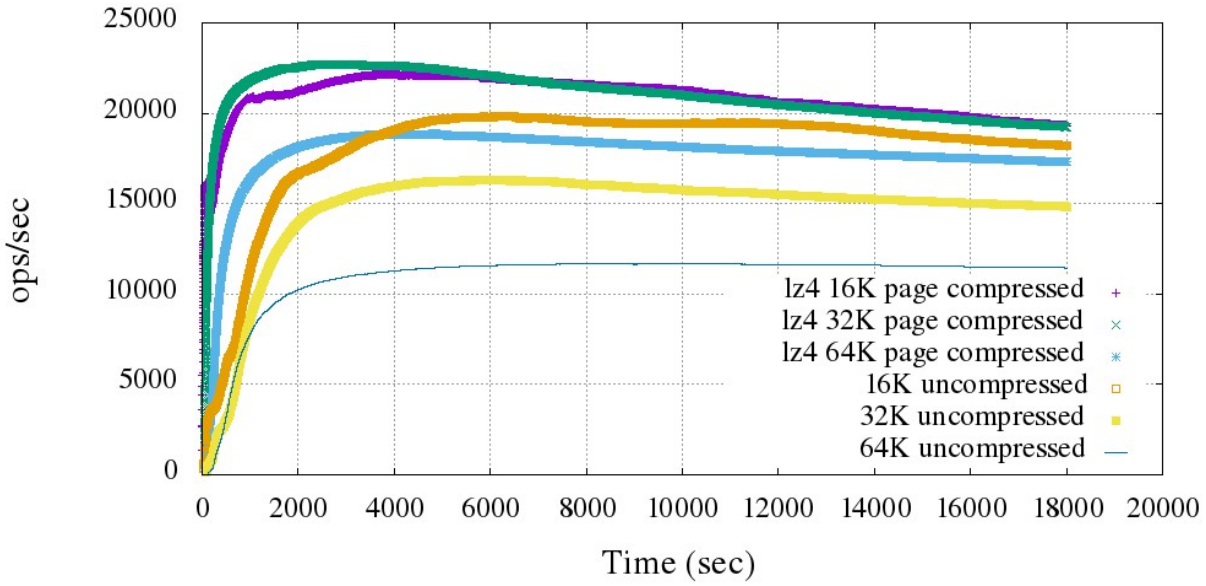


Figure 15: The number of LinkBench operations per second over a 5-hour measurement phase

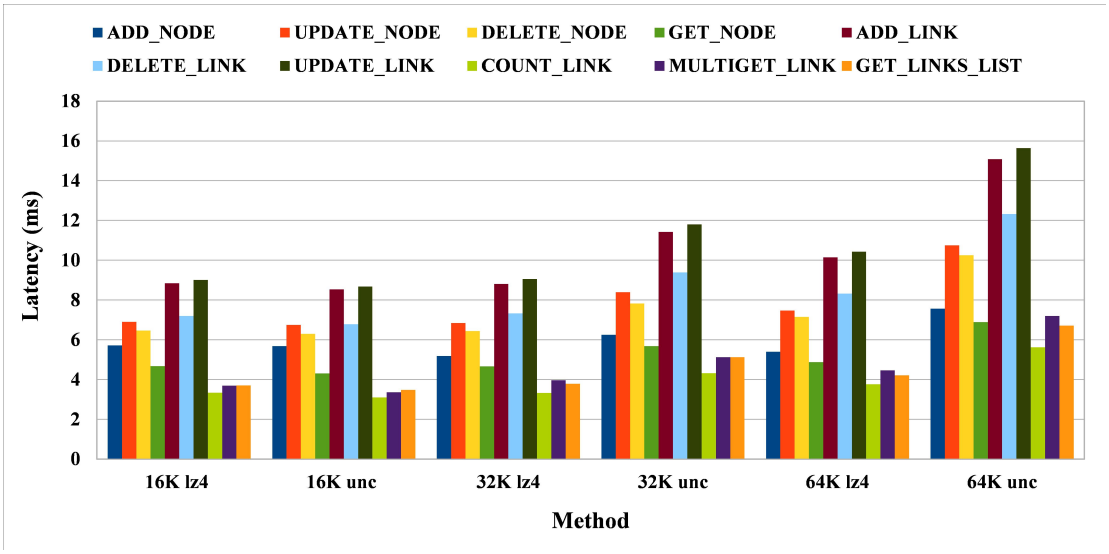


Figure 16: Latencies of LinkBench operations using lz4 compressed tables (lz4) and uncompressed tables (unc) with different database page sizes

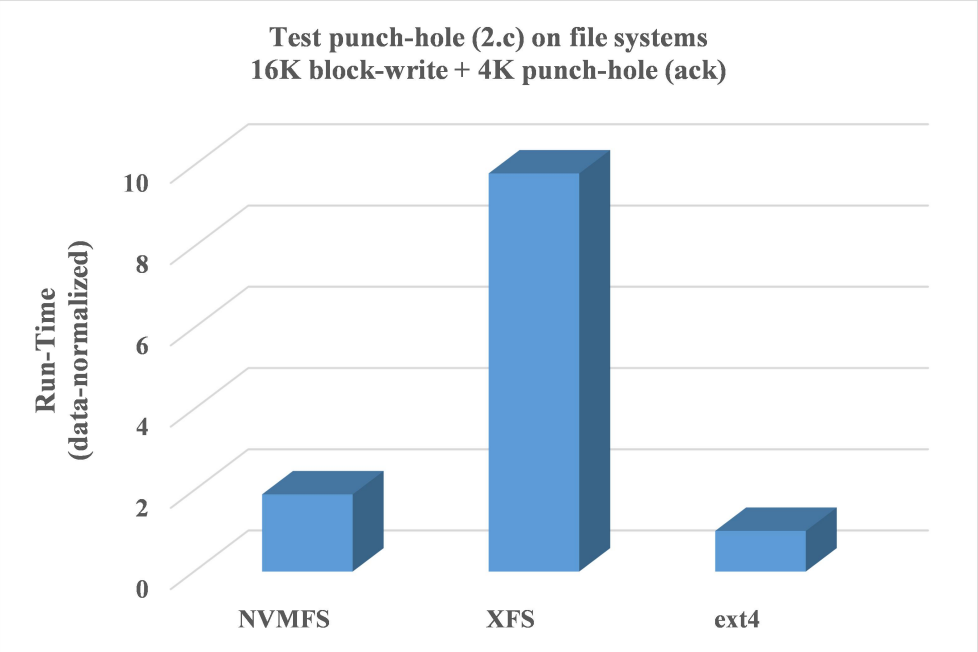


Figure 17: Trim operation: Punch hole comparison on different file systems

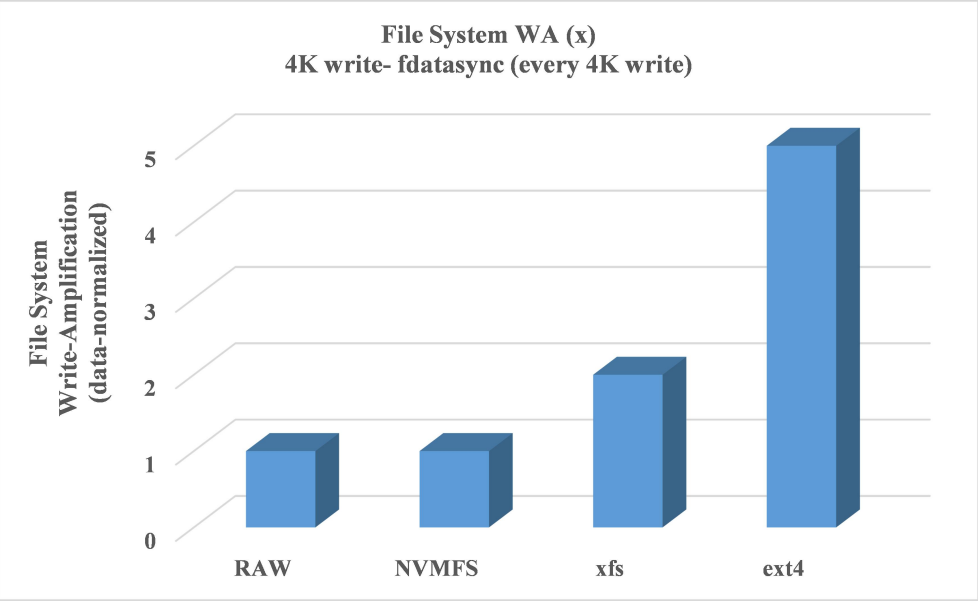


Figure 18: Write amplification on different file systems

different file systems using TPC-C like benchmark with 1000 warehouses, the buffer pool was sized to 75GB and the run time for the experiment is 1 hour. The throughput of Benchmark on NVMFS outperforms both XFS and EXT4 file systems. Not shown in the graph, as the file system ages with additional runs throughput on NVMFS is significantly better compared to XFS and EXT4 due to write amplication improvements and reduced garbage collection on the underlying flash storage.

The latency of generic operations like Drop Table have also been evaluated. The findings point to significant latency spike in drop table operations using NVM Compression technique on XFS and EXT4 file systems over similar tables uncompressed or the tables using legacy compression. Efficient usage of NVM primitives bring the performance improvement of NVMFS.

The evaluation of NVM Compression on various file systems provides insight into the workloads that require tuning on the respective file systems. Various micro benchmarks as described in the evaluation section can be used for rework and tuning as required for file systems on block devices that used FTL exposed NVM primitives for efficient file system operations.

6 Related Work

[18] found that storing data too densely in compressed pages incurs many future page splits similarly as in MySQL legacy row compression, which require exclusive locks. In order to avoid lock contention, [18] reduced page splits by sacrificing a couple of percent of space savings. Similar methods like adding padding can be used on MySQL. Such methods 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 showed that their 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 row compression method. NVM-compression achieved 2-7x better performance compared to same row compression method and improved storage efficiency by 19% and therefore seems to outperform their proposal.

Database compression technologies in general have been researched extensively earlier and one of best general introduction on this area is given in [24].

There has been active research on using NVM to reduce logging overhead, e.g., removing the disk and using NVM as the sole logging device [41, 11, 17, 19]. Similarly, there has been research on using NVM as virtual memory to optimize checkpoint writing [16]. In [23] Meng et. al. present experiences on flash-based database system FlashDB.

Another active research area on NVM is how to extend the endurance or lifetime of the devices [42]. In [15] Kaiser et. al. propose a method to extend lifetime of SSDs in databases using page overwrites. Similarly, the page size selection on databases using SSD storage has been researched in [33].

In [22] Marmol et. al. show how NVM primitives can be used to improve scalability and performance of key-value stores. In [1] Abadi et. al. show how compression can be used to improve the performance in column-oriented database systems. In [37, 38] Schwalb et. al. present how NVM primitives can be leveraged in database startup and crash recovery for in-memory databases.

Furthermore, there has been research on using new storage class memories as main memory [21, 30] and in main-memory databases [12]. [29] presents how database performance can be improved by using flash-based write cache. [13] proposes an AD-LRU method to improve buffer replacement for flash-based databases. Similarly, in [43] Yang et. al. propose an efficient buffer scheme for flash-based databases.

MariaDB is one kind of consistent database sytems. A comprehensive review on the state of the art of consistent databases can be found in [9][10]. A survery of performance on NoSQL database systems is presented in [2].

7 Conclusions

NVM-Compression is designed to combine the best of the 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. Various micro benchmarks as described in the evaluation section can be used for rework and tuning file systems using block devices exposing NVM primitives for efficiency improvement. The performance results are dramatic, with a performance close to or sometimes exceeding the uncompressed method 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™ [10] Percona™ [11] and Oracle™

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

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

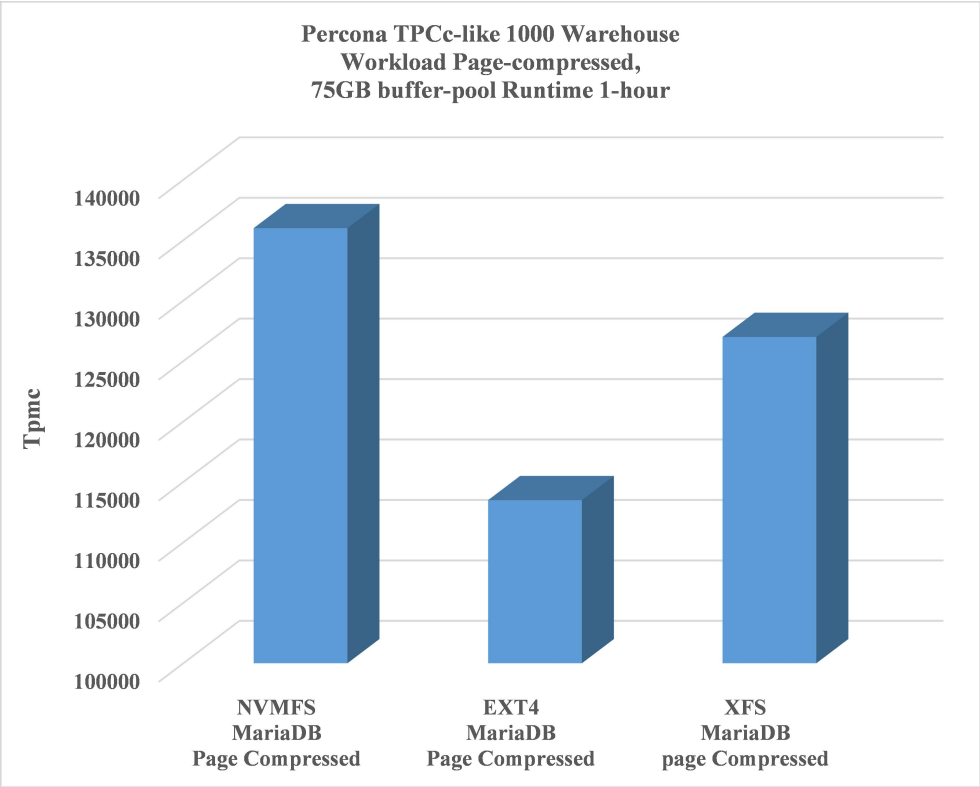


Figure 19: NVM compression on different file systems using TPC-C like benchmark

5.7.9(GA) [12].

MariaDB has been the reference implementation, and all development and analysis have been done on that code base. Other MySQL vendors have followed the suit and added their own implementation based on the work by MariaDB. The code for the MySQL storage engine implementation of NVM Compression is available as open source for all of the three distributions.

REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD Conference*, 2006, pp. 671–682.

[2] V. Abramova, J. Bernardino, and P. Furtado, "Which nosql database? a performance overview," *Open Journal of Databases (OJDB)*, vol. 1, no. 2, pp. 17–24, 2014. [Online]. Available: https://www.ronpub.com/ojdb/OJDB-v1i2n02_Abramova.html

[3] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a database benchmark based on the Facebook social graph," in *SIGMOD Conference*, 2013, pp. 1185–1196.

[4] D. Bartholomew, *"Getting Started with MariaDB"*. Packt Publishing, 2013.

[5] M. Callaghan, "Making InnoDB compression adaptive," https://www.facebook.com/note.php?note_id=10150345355665933, 2011.

[6] M. Callaghan, "The efect of page size on InnoDB compression," https://www.facebook.com/note.php?note_id=10150348315455933, 2011.

[7] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: The Next Generation of Ext2/3 Filesystem," in *Linux Storage & Filesystem Workshop*, 2007.

[8] Dhananjoy Das, "NVMFS," http://itblog.sandisk.com/in-a-battle-of-hardware-software-innovation-comes-out-on-top, 2014.

[9] M. M. Elbushra and J. Lindström, "Eventual consistent databases: State of the art," *Open Journal of Databases (OJDB)*, vol. 1, no. 1, pp. 26–41, 2014. [Online]. Available: https://www.ronpub.com/ojdb/OJDB-v1i1n03_Elbushra.html

[10] M. M. Elbushra and J. Lindström, "Causal consistent databases," *Open Journal of Databases*

[12] available at http://http://dev.mysql.com/downloads/mysql/

(OJDB), vol. 2, no. 1, pp. 17–35, 2015. [Online]. Available: https://www.ronpub.com/ojdb/OJDB_2015v2i1n02_Elbushra.html

[11] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proceedings of the 27th International Conference on Data Engineering*, April 11-16, 2011, Hannover, Germany, pp. 1221–1231.

[12] Y. Gottesman, J. Nider, R. I. Kat, Y. Weinsberg, and M. Factor, "Using Storage Class Memory Efficiently for an In-memory Database," in *Proceedings of the 9th ACM International on Systems and Storage Conference*, 2016.

[13] P. Jin, Y. Ou, T. Härder, and Z. Li, "AD-LRU: An efficient buffer replacement algorithm for flash-based databases," *Data Knowl. Eng.*, vol. 72, pp. 83–102, 2012.

[14] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," in *8th USENIX Conference on File and Storage Technologies*, 2010, pp. 85–100.

[15] J. Kaiser, F. Margaglia, and A. Brinkmann, ""extending SSD lifetime in database applications with page overwrites"," in *Proceedings of the 6th International Systems and Storage Conference*, 2013, pp. 1–12.

[16] S. Kannan, A. Gavrilovska, K. Schwan, and D. S. Milojicic, "Optimizing Checkpoints Using NVM as Virtual Memory," in *IPDPS*, 2013, pp. 29–40.

[17] A. Khekdar and V. Kumar, "Flash-based logging for database updates," in *CTS*, 2011, pp. 540–547.

[18] K. Lee, "Performance Improvement of Database Compression for OLTP Workloads," *IEICE Transactions*, vol. 97-D, no. 4, pp. 976–980, 2014.

[19] S.-W. Lee, B. Moon, C. Park, J. Y. Hwang, and K. Kim, "Accelerating In-Page Logging with Non-Volatile Memory," *IEEE Data Eng. Bull.*, vol. 33, no. 4, pp. 41–47, 2010.

[20] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "NVM duet: unified working memory and persistent store architecture," in *ASPLOS*, 2014, pp. 455–470.

[21] G. S. Lloyd and M. Gokhale, "Evaluating the feasibility of storage class memory as main memory," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 437–441.

[22] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, "NVMKV: A scalable and lightweight flash aware key-value store," in *6th USENIX Workshop on Hot Topics in Storage and File Systems*, Jun. 2014.

[23] X. Meng, L. Yue, and J. Xu, "Flash-Based Database Systems: Experiences from the FlashDB Project," in *DASFAA Workshops*, 2011, p. 240.

[24] N. J. Muller, "Database Compression Technologies," in *High-Performance Web Databases, Design, Development, and Deployment*, 2001, pp. 705–716.

[25] S. Oikawa, "Towards New Interface for Non-volatile Memory Storage," in *PECCS*, 2014, pp. 174–179.

[26] S. Oikawa and S. Miki, "File-Based Memory Management for Non-volatile Main Memory," in *COMPSAC*, 2013, pp. 559–568.

[27] S. Oikawa and S. Miki, "Future Non-volatile Memory Storage Architecture and File System Interface," in *CANDAR*, 2013, pp. 389–392.

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

[29] Y. Ou and T. Härder, "Improving Database Performance Using a Flash-Based Write Cache," in *DASFAA Workshops*, 2012, pp. 2–13.

[30] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory," in *Proceedings of the International Conference on Management of Data*, 2016, pp. 371–386.

[31] X. Ouyang, D. W. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, ""beyond block I/O: Rethinking traditional storage primitives"," in *HPCA*, 2011, pp. 301–311.

[32] Y. Park and J.-S. Kim, "zFTL: power-efficient data compression support for NAND flash-based consumer electronics devices," *IEEE Trans. Consumer Electronics*, vol. 57, no. 3, pp. 1148–1156, 2011.

[33] I. Petrov, R. Gottstein, T. Ivanov, D. Bausch, and A. P. Buchmann, "Page Size Selection for OLTP Databases on SSD Storage," *JIDM*, vol. 2, no. 1, pp. 11–18, 2011.

[34] F. Raab, "TPC-C - The Standard Benchmark for Online transaction Processing (OLTP)," in *The Benchmark Handbook*, 1993.

[35] I. Rana, "InnoDB compression improvements in MySQL 5.6," https://blogs.oracle.com/mysqlinnodb/entry/innodb_compression_improvements_in_mysql, 2012.

[36] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: a lightweight, consistent and durable storage cache," in *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference*, 2012, pp. 267–280.

[37] D. Schwalb, M. Faust, M. Dreseler, P. Flemming, and H. Plattner, "Leveraging non-volatile memory for instant restarts of in-memory database systems," in *32nd IEEE International Conference on Data Engineering*, 2016, pp. 1386–1389.

[38] D. Schwalb, G. Kumar, M. Dreseler, A. S., M. Faust, A. Hohl, T. Berning, G. Makkar, H. Plattner, and P. Deshmukh, "Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory," in *Database Systems for Advanced Applications - 21st International Conference*, 2016, pp. 267–282.

[39] N. Talagala, "Native flash support for applications," http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120823_S304B_Talagala.pdf, 2012.

[40] R. Y. Wang and T. E. Anderson, "xFS: A Wide Area Mass Storage File System," in *Proceedings Fourth Workshop on Workstation Operating Systems*, 1993, pp. 71–78.

[41] T. Wang and R. Johnson, "Scalable Logging through Emerging Non-Volatile Memory," *PVLDB*, vol. 7, no. 10, pp. 865–876, 2014.

[42] J. Yang, N. Plasson, G. Gillis, and N. Talagala, "HEC: improving endurance of high performance flash-based cache devices," in *SYSTOR*, 2013, p. 10.

[43] L. H. Yang, J. Wang, Z. Huang, W. Gong, and L. Chen, "An Efficient Buffer Scheme for Flash-based Databases," *JCP*, vol. 6, no. 7, pp. 1307–1318, 2011.

[44] K. S. Yim, H. Bahn, and K. Koh, "A flash compression layer for SmartMedia card systems," *IEEE Trans. Consumer Electronics*, vol. 50, no. 1, pp. 192–197, 2004.

[45] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

AUTHOR BIOGRAPHIES

Dr. Jan Lindstrm is the principal engineer at MariaDB working on InnoDB storage engine and Galera cluster. Before joining SkySQL he was software developer for IBM DB2 and development manager for IBM solidDB core development. He joined IBM with the acquisition of Solid Information Technology in 2008. Before joining Solid in 2006, Jan worked on Innobase and spent almost 10 years working in the database field as a researcher, developer, author, and educator. He has developed experimental database systems, and has authored, or co-authored, a number of research papers. His research interests include real-time databases, in-memory databases, distributed databases, transaction processing and concurrency control. Jan has an MSc. and Ph.D. in Computer Science from the University of Helsinki, Finland.

Dhananjoy Das is Sr. Architect at parallel machines working in new technologies in machine learning space. Before joining Parallel Machines, he was Principal Architect at SanDisk (formally Fusion-io, acquired by WD) working for more than 5 years in areas of advanced development, persistent memory, flash, distributed systems and application acceleration. His recent work includes the application of acceleration IO AMP stack to enhancing applications like MariaDB, MySQL and Cassandra for transactional efficiencies, write amplication reduction and flash endurance, providing improvement using FTL techniques for atomic updates and data compression methods. Prior to joining Fusion-io, He worked at NetApp for about 10 Years and designed infrastructure for fault tolerance, high availability for distributed systems across all storage platforms, worked on high speed interconnects and NVRAM infrastructure processing WAFL transactions. He is an active contributor to open source community and holds an MS. Computer Science from University of Pune, India.

Nick Piggin is now a software engineer at IBM. Before joining IBM, Nick Piggin worked for SanDisk (formally Fusion-io), and has interests in memory management, filesystems, and the Linux kernel, new storage and memory technologies. He has been working with these for the past several years.

Santhosh Kumar Koundinya is a software engineer at YellowBrick. Before joining YellowBrick, Santhosh worked at SanDisk, and was a core developer of the Non-Volatile Memory File System (NVMFS). NVMFS serves to connect applications to modern, high-speed persistent memories, and NVMFS v1.0 is availabe to customers.

Torben Mathiasen is now a principal engineer at YellowBrick and works on new technologies in the analytical space. He worked at Fusion-io and SanDisk where he has been evaluating software stacks for optimal performance and reliability for both internal R&D and customers. He has spent 11 years at Hewlett-Packard, architected firmware for the ProLiant BladeServer portfolio and worked as a Linux kernel developer. He also held a senior system architect position at Prevas, designed the next generation of embedded systems, which are used by customers from all over the world. He has made contributions to the Linux kernel in the areas like ethernet, PCI hotplugging, storage controller drivers and the SCSI stack.

Dr. Nisha Talagala is vice president of Engineering at Parallel Machines. Before joining Parallel Machines, Nisha was Fellow at SanDisk, where she worked on innovation in non volatile memory technologies and applications. Nisha has more than 10 years of expertise in software development, distributed systems, storage and I/O solutions, and non-volatile memory. She has worked as technology leader of server flash at Intel, CTO at Gear6 and Sun Microsystems. Nisha earned her PhD at UC Berkeley research clusters and distributed storage. Nisha holds more than 30 patents.

Dr. Dulcardo Arteaga currently works as software engineer at Parallel Machines. He obtained his Ph.D. in Computer Sciences at Florida International University (FIU) under the supervision of Dr. Ming Zhao in the Virtualized Infrastructure, Systems and Applications (VISA) Laboratory. Dulcardo obtained his Master of Science degree in Computer Science from FIU. Dulcardo's research focus is in block-level caching and virtualization.