An NVM-aware Storage Layout for Analytical Workloads

Philipp Götze
TU Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Stephan Baumann
Actian Germany GmbH
stephan.baumann@actian.com

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

***Abstract*—As DRAM is reaching its scalability limit, Non-Volatile Memory (NVM) technologies are moving more and more into focus to meet the requirements of modern database and Big Data systems. With this novel paradigm a lot of new opportunities but also challenges emerge which requires a rethinking of database system developers regarding the programming model and data placement. In this paper, we present a design of an NVM-aware storage layout for tables using a multi-dimensional clustering approach and a block-like structure to utilize the complete memory stack. Contrary to previous works, this approach focuses on analytical workloads. In a microbenchmark, its potential for scan and point queries is investigated. We show that our approach is able to outperform the considered competitors by up to one order of magnitude.**

I. Introduction

With the promising emergence of Non-Volatile Memory (NVM) new opportunities and challenges for data processing systems arise. In contrast to traditional databases, with NVMs direct persistence, near-DRAM performance, and byte-addressability only a single copy of the data needs to be present in the system. For instance, this allows a much simpler and faster crash recovery and avoids the detour via memory to disk and vice versa. In the past years, many researchers have already investigated this new paradigm. Especially persistent index structures emerge more and more frequently since main memory implementations do not go along with the properties of NVM.

Current persistent structures and systems, however, lack the feature to efficiently query records or columns on other attributes than the key [1]–[5]. Using a full-grown OLAP system would require a lot of installation and administration effort as well as additional storage costs. That is why a wide range of analytical applications would benefit from an NVM-enabled storage layout for a table. Our intention is to use this table at a later stage within a transactional stream processing system [6]. It is supposed to represent a persistent state on which both continuous as well as ad-hoc queries can be executed simultaneously. In this case, the key used for indexing is most likely the timestamp of the tuple. This is not very convenient for analytical workloads which often use other attributes than the key. The contributions of this paper are twofold as we strive to solve the following problems:

- Which data structures are necessary to utilize the memory stack (RAM, NVM, disk) for NVM-aware tables?
- How can we accelerate analytical queries on such tables even on non-key attributes?

Hence, the main issue tackled in this paper is to find a suitable storage layout with efficient data placement for analytical workloads under the presence of NVM. For that, we have designed a table based on multi-dimensional clustering using a block-like structure. In a microbenchmark, we compare the performance of our approach[1] with other open-source NVM-based data structures. Thereby, we focus on scan and point queries (collectively referred to as lookup) to demonstrate the potential of our approach.

II. Background

A. NVM characteristics

The general motivation behind NVM in contrast to DRAM is the better scalability, density, economic properties (no refresh power), and direct persistence without detours over memory. In comparison with flash, NVM offers the advantages of providing near-DRAM latencies, being byte-addressable and thus allowing fine-grained control. By combining the advantages of both technologies, the large performance gap between them is alleviated. Although most of the NVM technologies are not available on the market, yet, they present several attractive features, which are missing in either DRAM or flash storage. Of course, these technologies also have some drawbacks. The most important ones for our intentions are the novel programming challenges and the read-write asymmetry in terms of latency, which we have considered during our design.

The best-known representatives of the NVM technology are Phase-Change Memory (PCM) [7], Spin Transfer Torque RAM (STT-RAM) [8], and memristors (also referred to as RRAM or ReRAM) [9]. They promise latencies close to DRAM and a higher endurance than flash but still lower than DRAM. In Table I the properties of these technologies are summarized. The read-write asymmetry of the NVM technologies is easily recognizable, which ranges from five to ten times slower writes than reads. In addition, writes are not only slower but also cause cell-wearing reducing the device's lifetime. Another important feature is the density of these technologies as DRAM is reaching its limit.

[1] https://dbgit.prakinf.tu-ilmenau.de/code/PTable

TABLE I
CHARACTERISTICS OF CURRENT MEMORY TECHNOLOGIES [2], [7], [8], [10]–[12]

	Cell size	Latency r/w	Endurance
NAND	$4-6F^2$	$25/500\mu s$	10^4-10^5
DRAM	$6-10F^2$	$50/50ns$	$>10^{15}$
PCM	$4-12F^2$	$60/400ns$	10^7-10^9
STT-RAM	$6-50F^2$	$10/50ns$	$>10^{15}$
Memristor	$4-10F^2$	$10/50ns$	10^{11}

In [13] Oukid et al. have identified the main programming challenges for NVM such as persistent memory leaks, partial writes or recovery. Currently, we rely on Intel's Persistent Memory Development Kit (PMDK)[2], which already addresses most of the mentioned issues. The issues we are focusing on are the read-write asymmetry, data placement, and to some extent data recovery. Overcoming the asymmetry is one of the reasons why we chose a clustered (BDCC) block structure, which we describe in Section IV.

B. Bitwise Dimensional Co-Clustering

BDCC [14], provides a multi-dimensional clustering framework including data structures, data access, and processing techniques that have proven to be highly beneficial for analytical workloads. In the original work, the focus was on disk access. However, BDCC has proven to provide efficient access to millions of small groups and this way can be relevant for NVM with its capability of byte addressing data. At the same time, its support for multiple dimensions enables efficient query processing for more than just the key attribute(s). Being developed for column stores and, thus, decoupling indexing from clustering it can easily be transferred to main memory where data offsets can be calculated rather than relying on an additional translation layer.

BDCC, in a nutshell, stores similar tuples close to each other based on common properties mapped to an artificial clustering key. Such a BDCC key value is created by bit interleaving binary representations of the chosen clustering dimensions. In order to efficiently point and range query dimensions, each dimension is sorted and mapped (according to this sort order) to a finite sequence of integers starting from 0. Consequently, multiple lookups on dimension attributes can be executed as a set of bit operations on the clustering key, proving very efficient selection pushdown and, in combination with out of order data retrieval, close to zero cost data reordering based on the various dimension sort orders. For bit-interleaving of the BDCC clustering key, different strategies such as round-robin or major-minor, but also any other interleaving, are possible and an optimal choice may actually depend on the data set and/or the workload.

The term co-clustered in this context refers to the contiguously multi-dimensional storage of tuples across relations and based on their foreign key relationships. In our case we consider just a single table, thus this focus is of no importance in this work. Although we have chosen BDCC due to its proven flexibility and efficiency, our design will not be limited to this clustering approach.

[2]http://pmem.io/pmdk

III. RELATED WORK

There are already several publications seeking for appropriate data structures on NVM. FOEDUS [2] covers multiple aspects of designing an OLTP engine on future hardware. Their data structures and algorithms are designed to be massively parallel under the assumption of the existence of NVM. The implementation of FOEDUS is the first and only major implementation to cover the aspects of many cores in combination with NVM. SOFORT [3] is a hybrid NVM-DRAM transactional storage engine supporting transactional as well as analytical workloads. The tables are organized column-wise and the main part of the data is kept in a read-optimized structure to accelerate OLAP performance. Additionally, a write-optimized delta storage is used for OLTP workloads which is merged periodically into the main part. Although the authors state that the engine is intended for mixed workloads, they rather focused on OLTP and recovery performance. Our focus, on the other hand, is on analytical queries.

Apart from these data processing engines, a couple of persistent index structures for NVM have been published. Venkataraman et al. [5] propose a single level storage hierarchy and present the idea of consistent and durable data structures. Their main use case is the B$^+$-Tree, which is also addressed in depth by Chen et al. in [1]. This approach focuses on optimizing the write performance for B$^+$-Trees in the presence of asymmetric read-write latency and the issue of wearing memory cells. In [4] the authors present a hybrid solution for a persistent and concurrent B$^+$-Tree, namely FPTree. For that, they hold all leaf nodes within persistent memory and inner nodes are placed in DRAM. They strive to achieve nearly the same speed as a DRAM-based B$^+$-Tree. For that, they use various techniques to accelerate tree operations and recovery. The crucial part is the usage of fingerprints in the leaf nodes to reduce the number of keys probed when searching for a key. Since these are index structures, they offer no analytical operations other than basic point queries on the key. Our structure, on the other hand, allows efficient arbitrary lookups for both key and non-key attributes.

In addition to the NVM-specific solutions, other approaches such as bitmaps, zonemaps, and Column Imprints [15] exist, which also aim to accelerate analytical queries. Since our table is supposed to be usable for both streaming and ad-hoc processing in the future, it is similar to Lang's et al. Data Blocks [16] which are intended for hybrid OLTP and OLAP database systems. They use a PAX-like structure where the blocks are horizontally partitioned per attribute in so-called mini pages and all columns of a tuple remain in the same block. These blocks are further enhanced with small materialized aggregates (SMAs) [17], a light-weight index, and compression to achieve much better cache performance and accelerate table scans.

IV. Analytical Table Structure

A. Access patterns

As we focus on analytical workloads it is important to discuss the expected access patterns of such workloads beforehand. Typically, there will be many more read than write operations. We mainly aim to support fast range scans and random point queries. Since our table represents not just a key-value store, it has to allow for mixed attribute ranges and queries on attributes other than the key. Additionally, it needs to be optimized for columnar accesses where the user is only interested in specific attributes.

B. Objectives

Justified by the NVM characteristics and the access patterns for analytical workloads described before, an appropriate data structure is required. We aim to achieve a better performance than typical table structures in modern database systems which are using a combination of a persistent table stored on disk and a volatile version hold in memory. Simultaneously, we strive to reduce writes to NVM whenever possible when operating on these tables to overcome the read-write asymmetry and to avoid early cell wearing. This is especially important for the data manipulation operations, namely insert, delete, and update as well as the general data organization. To achieve efficient OLAP performance a read-optimized data structure is necessary. To provide a high query performance as well as the possibility to swap cold data in case of an NVM space shortage, the approach covers a three-layer memory hierarchy. This is especially important as we expect the first generations of NVM on the broad market to have higher latencies than DRAM and lower capacities than flash (also considering the cost per GB).

C. Block structure

In order to utilize NVM and disk together for persistence, we decided for the common denominator of a block-like structure. This also achieves data locality, which in turn is beneficial for exploiting caches and prefetching mechanisms. Because the bytes of a block in NVM can be accessed individually this does not present a restriction. Similar like Data Blocks [16], we utilize a PAX-like structure and enhance it with SMAs. For the time being, the blocks do not make use of a light-weight index and compression but it could be part of future work. In Figure 1 our layout of such a block is shown.

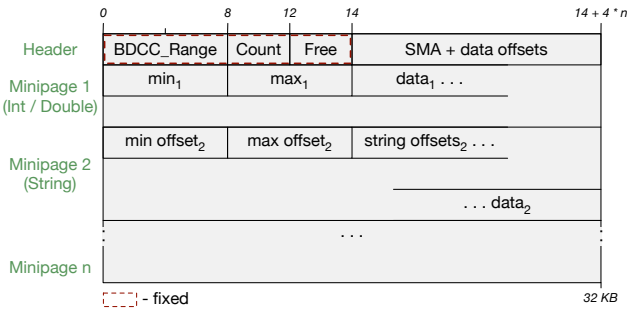


Fig. 1. Clustered block structure

The first fields represent the header consisting of the clustering key range, tuple count, and free space of this block. After that, the offsets to the attributes SMAs and data values are noted which are calculated based on the tuple's number and types of attributes. The body then is composed of the mini pages for each attribute. Currently, three basic types are supported: integer, double, and string values; where the string type has to be organized differently due to its variable length property. For the size of each cluster, we have chosen 32 KB for the time being as this is the minimum size limit before reading from solid state drives becomes inefficient [18]. Apart from that, this matches with today's typical L1 cache sizes which can be useful when accessing the same block multiple times.

D. Clustering

In OLAP queries, often other attributes besides the key are used for lookups. Instead of holding additional secondary indexes that would cost more write performance within NVM or recovery effort when using RAM, our approach is to cluster similar tuples together within the blocks. To achieve this, our approach is based on BDCC to allow multi-dimensional accesses and accelerate analytical workloads. We could have used also other clustering methods, but BDCC has proven itself quite effective [14] and is the most flexible approach. Through fast bitwise operations, one can massively accelerate queries. As a side effect, compression is also made possible since similar values are stored close to each other. The clustered blocks are sorted by the BDCC ranges and form a linked list. However, their respective contents are heap organized. This results in less writes for re-sorting and because of NVM's fast random access, there is no big disadvantage for scans.

E. Storage layout

In Figure 2, this idea of a table based on Data Blocks and BDCC within the memory hierarchy is sketched. The structure consists of three components: the table metadata, an index structure, and the actual table data distributed among NVM and disk. The metadata is the entrance object to the persistent area and holds a persistent pointer to both the index structure as well as the first block. In addition, the schema and clustering information is stored here and needs to be specified by the user on table creation. For the index, no specific structure is fixed. Currently, it is a basic NVM-persistent B^+-Tree version. An alternative could be, e.g., a volatile variant which in turn has to be recovered in case of a failure. The value of an index entry here is a specific class which we call `PTuple` that consists of a persistent pointer to a block plus the offsets of all attributes. Frequently used persistent pointers (or paths) are also cached to avoid pointer chasing. As for now, a persistent pointer is an NVM-only pointer and a transcending variant for disk and NVM will be part of future work. A simple solution would be to use a boolean field combined with a union structure (for the different pointer types). A further point of discussion is the distinction between hot and cold blocks, i.e. whether they should differ in the structure or just the location. Moreover, one could keep the metadata and the links of the data nodes

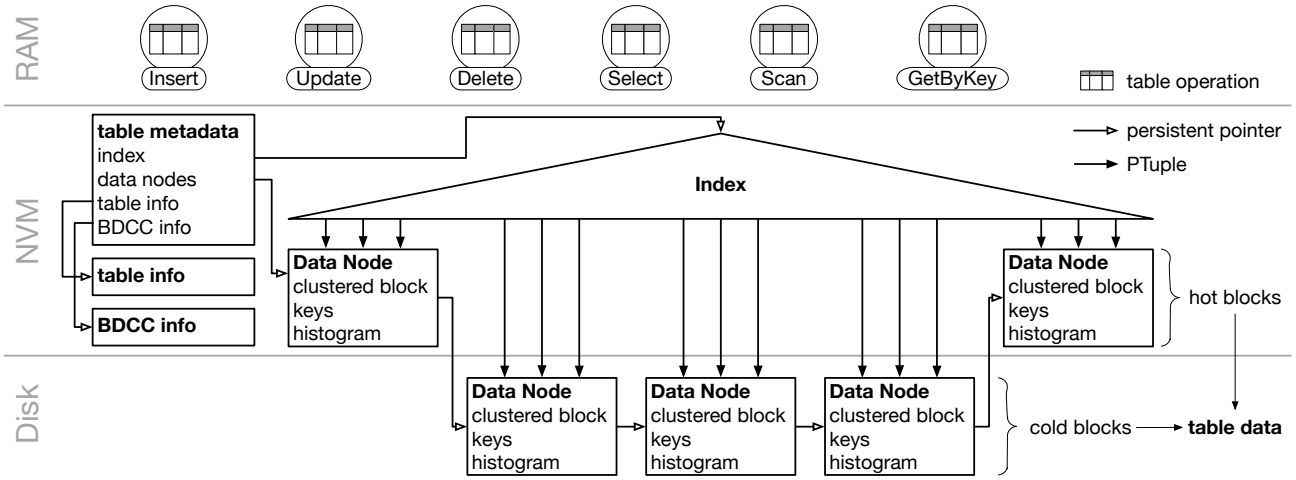


Fig. 2. Storage layout of a table for analytical workloads

always within NVM and only move the 32 KB arrays as soon as necessary. In the current state, however, all structures are stored in NVM. Therefore, and due to the usage of atomic transactions, no recovery is required.

F. Operations and optimization

Our table currently supports the basic data manipulation operations, namely insert, delete, and update, as well as fundamental analytical operations such as range scans, point queries on the key, and other selections with arbitrary predicates. Intermediate results are kept in DRAM which is why these operations are located at this layer in the figure. When appending a new tuple its BDCC value is calculated and it gets inserted in both the appropriate block covering this value and the index. This is especially useful as the data is organized by the key among other specified dimensions and thus allows fast appends. This process is additionally accelerated because the entries within a block are unsorted. Due to NVM, the tuples can be directly persisted and read later without the detour via memory to disk and vice versa. A block is split as evenly as possible as soon as it is full. This is done based on the BDCC range distribution using a histogram. The index is mainly used for efficient single tuple accesses or custom predicate scans. With our clustered block structure we want to achieve especially fast and robust range scans. For that, the number of blocks necessary to access must be limited. If the attributes used within the predicate are part of the clustering they can be masked with BDCC and compared with the block headers. For other attributes, the SMAs can be used instead. In this case, we do not use the index but a separate block iterator to collect all candidates and prune non-intersecting blocks. To further accelerate this process, one could also make use of an additional block index, such as a binary search tree.

G. Summary

In summary, the NVM properties were considered as follows. In order to counteract the asymmetry, we have made various design decisions. The most essential are the use of clustering instead of secondary indexes and the heap organization of the block contents. Intermediate results are stored in DRAM to save unnecessary NVM write operations. There is also the possibility of compression, which trades the number of bytes to write for a longer processing time. Due to the direct persistence of NVM, the use of atomic transactions, and by keeping all necessary data in NVM, no recovery is required.

V. Evaluation

The following microbenchmark is intended to highlight the potential of our approach over other NVM-based solutions and the commonly used DRAM + disk approach. Here, we concentrate on reading performance and analytical workloads. We expect faster lookup operations than our competitors, especially for queries on non-key attributes. As comparative systems for the evaluation, we considered open-source NVM-based data structures such as pmemkv[3], a ctree[4], and our own B^+-Tree version based on NVM. This tree is currently also used as index within our table implementation. Apart from these NVM-based competitors, we have further added an upper and lower baseline. Due to its wide usage in the research community as a persistent key-value store, RocksDB[5] was added as a disk-based reference system. It is important to point out that RocksDB uses the disk (SSD) for persistence and simultaneously keeps a transient version of the data in memory. Finally, we also included a DRAM-based B^+-Tree as a reference transient implementation. We presume the performance of the NVM-based approaches to be located between these two baselines.

A. Setup

For the implementation, we currently rely on emulation of NVM with the help of memory mapping. This means that a part of the memory is permanently reserved and treated as persistent memory. The mounted file system on this region is ext4 with DAX[6] support. As a result, load and store operations can directly access the persistent memory without the detour via the OS cache (paging). A disadvantage of this type of

[3]https://github.com/pmem/pmemkv
[4]Part of Intel's PMDK examples
[5]http://rocksdb.org/
[6]https://www.kernel.org/doc/Documentation/filesystems/dax.txt

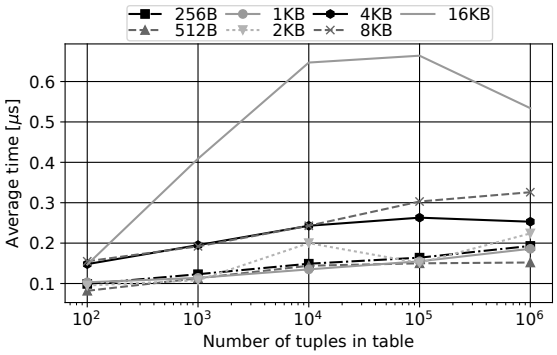


Fig. 3. Point queries for varying index node sizes

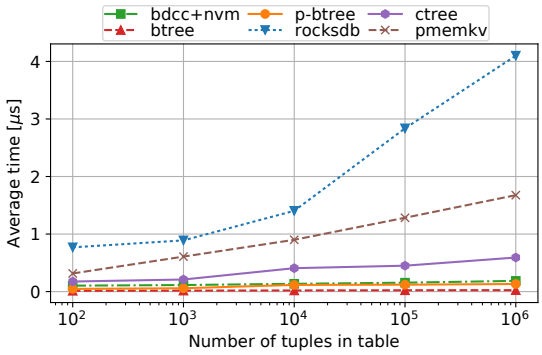


Fig. 4. Point query

emulation is that the read-write latency, unlike the expected NVM properties, is symmetric. We aim to utilize the real hardware as soon as it is available on the market. To address the persistent memory we use the C++ bindings of PMDK.

The tests of our microbenchmark are executed on a single node with an Intel Core i7-7500 processor running at 3.50 GHz. Each of the four cores features a 32 KB first level data and instruction cache as well as a 256 KB L2 cache. Furthermore, they all share a 4 MB L3 cache. The machine provides 16 GB of DRAM from which 4 GB are mapped as persistent memory. The DRAM latency measured with the Intel Memory Latency Checker[7] is 90ns.

We focused on comparing read-only settings, namely scan and point queries. For the scan queries, the selectivity was varied and for the point queries different table sizes were chosen to test for scalability. Since we strive for structured data, the values are always tuples of four elements (<int, int, string, double>) and an integer key (same as the first attribute). Most of the measurements do not contain any serialization, deserialization, or dereference overhead. The only exception occurs during the non-key queries for some solutions, such as RocksDB. There, the deserialization was necessary to access the individual attributes of the value. All tests were compiled with GCC 7.2.1.

B. Point query

The following point queries are all based on the key. In a preliminary experiment, we varied the node sizes of our index structure to determine the best performing configuration. The results are shown in Figure 3. The minimum size for a node is 256 Bytes. This is due to the size of the meta structure and the PTuple with four elements. The diagram shows that the best size ranges between 256 and 1024 Bytes. Some lines, e.g., for 2 KB and 16 KB, are not quite linearly despite multiple runs. This is probably because for some table sizes a better index state is created. We decided for a node size of 1 KB as the performance difference is only marginal. In addition, this avoids too many allocations and splits during the insert when using smaller sizes.

The node size of 1 KB is applied to our own approach and the NVM-based B$^+$-Tree. For the transient version, 4 KB has revealed the best performance. This is most likely due to the OS cache using a 4 KB page size. Figure 4 presents the comparison between the competitor systems. As it can be seen, our approach outperforms RocksDB, pmemkv, and the ctree. RocksDB and pmemkv, moreover, do not seem to scale as good as the other data stores. Since the persistent B$^+$-Tree version is used within our table, a similar performance is achieved. The difference is that the BDCC approach uses a PTuple, whereas the plain tree uses C++ tuples. The transient B$^+$-Tree (14-25 ns) clearly outperforms the other solutions. However, this obviously does not provide the desired persistence.

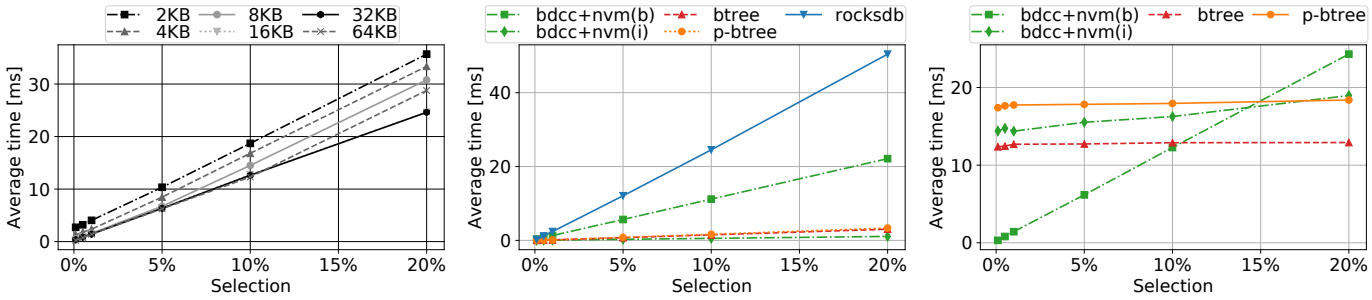
C. Range scan

For the range scan tests, we considered typical key based scans as well as scans using non-key attributes. Especially the latter experiment is of essential interest. As mentioned before, our future use case shall exploit this structure to represent persistent states in a transactional stream processing system. The key is certainly the timestamp and, thus, usually not the target of the range predicate. In this benchmark, we used the first (integer) and fourth (double) field for the range predicates. The ranges were chosen in a way that they overlap by 50 percent and result in the same selection as the key based scans. All scan operations were executed on the structures with 1 million tuples inserted beforehand.

As stated before, the block size was chosen to be 32 KB due to efficient SSD I/O operations and the L1 cache size. Nevertheless, we evaluated the best block size for an NVM only (for persistence) setting. Thus, in another preliminary experiment, the block size was varied while running non-key range scans. Figure 5(a) shows the average execution time of this query. Interestingly, the scans consistently perform best for 16 KB and 32 KB - in the figure both lines overlay each other. Consequently, a 32 KB block size is a good choice for this workload on both SSDs and NVM.

Considering range scans, our approach utilizes two types of iterators. The first is the block iterator using a preceding pruning mechanism as described in the previous section. The second is the iterator of the underlying index. In the following experiments, the competitor systems were additionally included. pmemkv does not provide a scan operation or an iterator, therefore, only its point query performance could be measured. RocksDB and the ctree were also omitted in the graphics, as they performed a magnitude worse than the other

[7]http://www.intel.com/software/mlc



(a) Varying block sizes using non-key attributes (b) Using the key (c) Using non-key attributes

Fig. 5. Range scan on a table of one million tuples (type: <int, int, string, double>)

solutions. The results of the key and non-key range scans are visualized in Figure 5(b) and Figure 5(c), respectively.

Although our focus was especially on queries with non-key attributes, the BDCC based table implementation performs similar or even better than the competitor systems. As the ctree does not support range scans, all the scan queries degenerate to full-table scans ($\approx 105ms$). The same applies for RocksDB ($\approx 275ms$) and the B+-Trees in the case of scans on non-key attributes. Our structure using the block iterator performs within the same range w.r.t. non-key execution time as the B^+-Tree and, thus, provides persistence and DRAM performance simultaneously. However, the execution time increases linearly with the selection percentage. This is because with an increasing tuple range the pruning step can exclude fewer blocks. In this case, the index iterator gets faster still achieving the same performance range as the transient and persistent B^+-Tree. Thus, the choice of iterator highly depends on the table size and selectivity. For larger volumes (>1M tuples) and a low selection amount, the block iterator is most likely to be preferred over the index utilization. It is expected that with a larger table size, the intersection point will move further to the right. This is because the index structures take longer for a complete scan, regardless of the selection. Consequently, the system would benefit from a cost-model which autonomously decides on the appropriate iterator. All in all, it has been shown that our approach can keep up with and even surpass our competitors in all aspects considered.

VI. Conclusion

In this paper, we introduced a clustered NVM-aware storage layout for analytical workloads. The results of our microbenchmark show that most of the considered persistent data stores lack the possibility to efficiently access entries based on non-key attributes. On the other hand, our storage layout for a table has proven to be very suitable for modern analytical performance requirements. Even for key-based queries, our structure performs similarly to its competitors and partly even outperforms them. In the near future, the intention is to further extend the microbenchmark to an exhaustive NVM based data store benchmark for OLTP as well as OLAP workloads. Furthermore, the presented solution still has a lot of optimization potential. For instance, BDCC could be exploited with more bitwise operations and various clustering options. Moreover, the data placement and node sizes within specific scenarios need to be evaluated in more detail. Finally, the support for concurrent access is currently missing and the structure should be enhanced with, e.g., MVCC.

Acknowledgement

This work was partially funded by the German Research Foundation (DFG) in the context of the project "Transactional Stream Processing on Non-Volatile Memory" (SA 782/28) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037).

References

[1] S. Chen and Q. Jin, "Persistent B+-Trees in Non- Volatile Main Memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, 2015.
[2] H. Kimura, "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM," in *SIGMOD*, 2015, pp. 691–706.
[3] I. Oukid, D. Booss *et al.*, "SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery," in *DaMoN*, 2014, pp. 8:1–8:7.
[4] I. Oukid, J. Lasperas *et al.*, "FPTree: A Hybrid SCM- DRAM Persistent and Concurrent B-Tree for Storage Class Memory," in *SIGMOD*, 2016, pp. 371–386.
[5] S. Venkataraman, N. Tolia *et al.*, "Consistent and Durable Data Structures for Non-Volatile Byte- Addressable Memory," in *USENIX*, 2011, pp. 61–75.
[6] I. Botan, P. M. Fischer *et al.*, "Transactional Stream Processing," in *EDBT*, 2012, pp. 204–215.
[7] B. C. Lee, E. Ipek *et al.*, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009, pp. 2–13.
[8] M. Hosomi, H. Yamagishi *et al.*, "A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: Spin-RAM," *IEDM Tech. Dig*, vol. 459, 2005.
[9] D. B. Strukov, G. S. Snider *et al.*, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.
[10] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE TPDS*, vol. 27, no. 5, pp. 1537–1550, 2016.
[11] H. Saadeldeen, D. Franklin *et al.*, "Memristors for Neural Branch Prediction: A Case Study in Strict Latency and Write Endurance Challenges," in *CF*, 2013, pp. 26:1–26:10.
[12] T. Wang and R. Johnson, "Scalable Logging through Emerging Non-Volatile Memory," *PVLDB*, vol. 7, no. 10, pp. 865–876, 2014.
[13] I. Oukid, D. Booss *et al.*, "Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems," *PVLDB*, vol. 10, no. 11, pp. 1166–1177, 2017.
[14] S. Baumann, P. A. Boncz, and K. Sattler, "Bitwise Dimensional Co-Clustering for Analytical Workloads," *VLDB J.*, vol. 25, no. 3, pp. 291–316, 2016.
[15] L. Sidirourgos and M. L. Kersten, "Column Imprints: A Secondary Index Structure," in *SIGMOD*, 2013, pp. 893–904.
[16] H. Lang, T. Mühlbauer *et al.*, "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation," in *SIGMOD*, 2016, pp. 311–326.
[17] G. Moerkotte, "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing," in *VLDB*, 1998, pp. 476–487.
[18] S. Baumann, G. de Nijs *et al.*, "Flashing Databases: Expectations and Limitations," in *DaMoN*, 2010, pp. 9–18.