SAP HANA Adoption of Non-Volatile Memory

Mihnea Andrei
Christian Lemke
Günter Radestock
Robert Schulze
Carsten Thiel

Rolando Blanco
Akanksha Meghlan
Muhammad Sharique
Sebastian Seifert
Surendra Vishnoi

Daniel Booss
Thomas Peh
Ivan Schreter
Werner Thesing
Mehul Wagle

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
{first_name}.{last_name}@sap.com

Thomas Willhalm
Intel Deutschland GmbH
thomas.willhalm@intel.com

ABSTRACT

Non-Volatile RAM (NVRAM) is a novel class of hardware technology which is an interesting blend of two storage paradigms: byte-addressable DRAM and block-addressable storage (e.g. HDD/SSD). Most of the existing enterprise relational data management systems such as SAP HANA have their internal architecture based on the inherent assumption that memory is volatile and base their persistence on explicit handling of block-oriented storage devices. In this paper, we present the early adoption of Non-Volatile Memory within the SAP HANA Database, from the architectural and technical angles. We discuss our architectural choices, dive deeper into a few challenges of the NVRAM integration and their solutions, and share our experimental results. As we present our solutions for the NVRAM integration, we also give, as a basis, a detailed description of the relevant HANA internals.

1. INTRODUCTION

The SAP HANA data platform [5] (hereafter referred to as 'HANA') is a next-generation database, built from scratch by SAP. Its architecture was originally based on the advent of modern hardware, as large main memory capacities and high core counts. Continuous early adoption of game changing hardware innovation, like new vector instructions or hardware transactional memory, remains a powerful driver of HANA's technical roadmap. The Persistent or Non-Volatile or Storage Class Memory (hereafter referred to as 'NVM' or 'NVRAM') is one such game-changing new hardware technology that promises to combine the best of both worlds, namely memory and storage.

As seen in Figure **1**, computing applications so far have been organizing their data between two storage tiers: memory and disk.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

With the advent of NVRAM technologies like 3D XPoint [7], PRAM [3], MRAM [1], Memristors [19], the dichotomy between memory and storage is about to change with the introduction of a third tier.

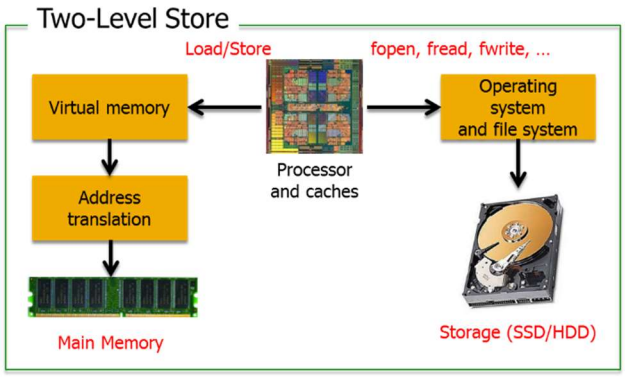

Figure 1: Dichotomy of storage

NVRAM is an emerging class of memory storage devices offering a DIMM form-factor. Hence, it can be treated by the CPU as RAM, not as a block device: it is byte-addressable, directly accessible using load/store instructions, and covered by CPU cache-line handling, including cross-socket cache coherency. The device latency is expected to be close to that of DRAM and its bandwidth lower than that of DRAM. The density, durability and economic characteristics however match that of existing block-based storage media. Due to these attributes, NVRAM can potentially benefit applications that deal with big data analytics, in-memory databases, high availability systems, etc.

Most of the existing relational DBMSs have their internal architecture based on the assumption that memory is volatile and that memory is a scarcer resource than disk is. If such DBMSs properly amend this assumption, they can vastly benefit from the offerings of NVM. The specific features of HANA and NVM make this adoption both promising and challenging. In this paper, we present the salient elements of the early adoption.

In-memory DBMSs such as HANA have always dealt with the trade-off between in-memory vs. durable storage as shown in Figure 2. Having data in DRAM provides faster read and write

performance, but all DRAM data is lost once the power is turned off. In order to offer ACID transactions and data durability, HANA relies on write-ahead redo logging and writes data to a non-volatile storage device which supports slow bulk data transfer as blocks.

Likewise, as an in-memory database, HANA is bound by the per-node available memory limit and scales-out to go beyond it. A technology promising a substantial increase of per-node memory volume at a reasonable price, such as NVRAM, is thus specifically interesting.

Finally, as most of the data structures are processed in DRAM, restarting an in-memory DBMS involves reloading the tables/columns and rebuilding the auxiliary data structures. This strong correlation with the data volume results in a substantial restart time for the DBMS . For instance, it may take up to an hour to load Terabytes of data into main memory, thus increasing the system downtime. As NVRAM is byte-addressable and writes to NVRAM are durable, a database designed to exploit that can access data directly in NVRAM after a restart, without having to reload it from disk into DRAM. From the end-user perspective, such a system can significantly boost its operational performance in terms of quickly bringing the business critical relational tables online after an unplanned down time or even after planned system upgrades. Note that one hour of down-time per year reduces system availability below 99,99% ([12]). Such use of NVRAM is thus a step forward to increase the system availability.

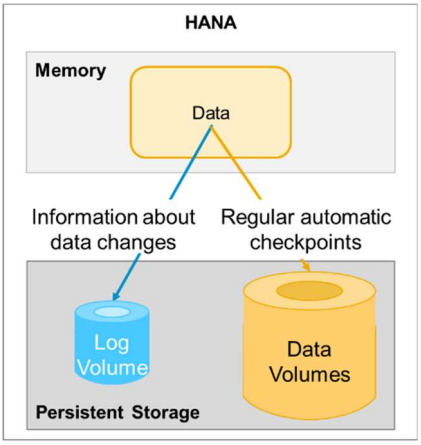

Figure 2: HANA's usage of main memory vs disk storage

However, the adoption of NVM by a relational DBMS has quite a few open questions and challenges. Whether to use the DIMM form factor, i.e. NVRAM, as a larger and persistent RAM, as opposed to using the SSD form factor of NVM, i.e. as a faster disk. And how actually to do that, i.e. how to handle NVRAM mixed with DRAM in the DBMS virtual memory address space. What data structures to place in NVRAM. How to handle their persistent lifecycle. What abstraction to expose to the DBMS when integrating the NVRAM technology. Note that the above challenges are not specific to HANA. Other in-memory DBMS will also face them in adopting NVRAM.

To address such questions, this paper has the following contributions:

- a discussion of using the DIMM form-factor of NVRAM as preferable then the SSD one, and an architecture to adopt that in an in-memory DBMS like HANA
- an in-memory format for column data which is directly stored and used in NVRAM blocks
- the integration of NVRAM blocks and their stored items within the column store
- the integration of NVRAM blocks within the lower persistency layer
- and the experimental data to test our assumptions before the NVRAM final DIMMs are available.

Each contribution is summarized at the end of the section presenting it. Moreover, the sections describing the detailed NVRAM integration have the further contribution of describing in sufficient detail HANA's relevant internal structures and processing. We also stress where the contributions apply to a more general class of in-memory DBMSs, beyond HANA.

The paper has the following structure: in section 2, we present the HANA database. Then, we introduce the relevant features of the NVRAM technology in section 3. Section 4 gives the high level architectural elements of HANA's adoption of NVRAM. The following sections, 5, 6, and 7, dive into deeper details of the NVRAM impact on the column data format, column store, and persistency layer. We give our experimental results in section 8. We present the related work in section 9. Finally, section 10 concludes the paper.

2. THE SAP HANA DATABASE

The goal of HANA is the integration of transactional and analytical workloads within the same data management system. To achieve this, a columnar engine exploits modern hardware (multi-processing cores, SIMD, large main memory and processor caches), compression of database content, maximum parallelization in the database kernel, and database extensions required by enterprise applications (specialized data structures for hierarchies or support for domain specific languages).

As seen from Figure 3, in the transactional workload of the SAP business applications, more than 80% of all statements are read accesses [10]. The remaining data write operations consists mainly of inserts, a few updates, and very rare deletes. The analytical workloads are even more dominated by reads. This distribution is extracted from customer database statistics. In contrast, the TPC-C benchmark, that has been the foundation for optimizations over the last decade, has a higher write ratio (46%).

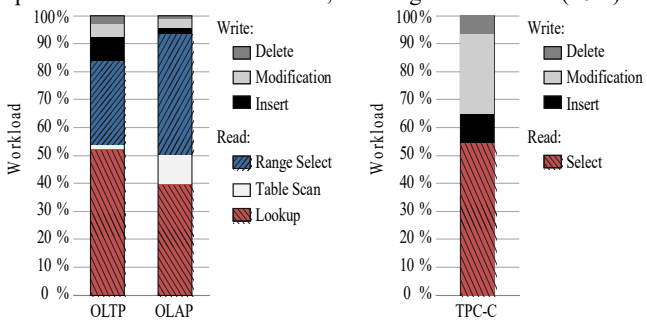

Figure 3: DBMS workload characteristics [10]

HANA's relational in-memory columnar store is tuned for such enterprise workloads. In-memory processing reconciles the traditionally row-oriented transactional database design with the innovative column-oriented design that is optimal for analytics. This allows both types of workloads to be executed on the same operational data. The trade-off favors the large majority of read accesses through the choice of a columnar database. The performance impact on the row-oriented smaller transactional workload is acceptable due to the fast in-memory access.

HANA optimizes its utilization of the available memory by implementing extensive compression technologies within its columnar store. Each column is at least dictionary-compressed whereas the value identifiers are again compressed using N-bit encoding. Depending on the data distribution, advanced compression techniques may be further applied as seen in Figure 4 (e.g. prefix, run-length, etc.).

Compression pays off best when data does not change, in which case the computational effort spent to compress data is leveraged over a longer duration. The data structures optimal for storing and processing compressed data are however not update-friendly. As an example, let us consider the dictionary. On one hand, a sorted dictionary is friendly to both query processing and memory consumption: it supports direct binary search, and does not require the overhead of a dictionary index. Also, range queries can execute directly on the integer value encodings, and thereby avoid the actual domain's value lookup. On the other hand, a sorted dictionary is not friendly to inserting new values since they do not come in order and the dictionary needs to be re-sorted/re-organized all the time. This example highlights a typical data processing dilemma seen as a conflict when optimizing for both the analytical and the transactional workloads.

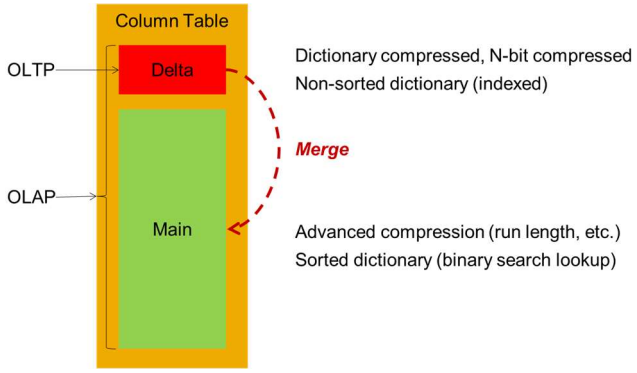

Figure 4: Data stores in SAP HANA

HANA resolves this conflict by using two *Table Fragments* per columnar table, with different store organizations: Main and Delta. Each Table Fragment has a *Column Fragment* for each table column. The Main fragment is reader-friendly: it contains most of the data, and it changes rarely; it uses sorted dictionaries, N-bit and other compression techniques. The Delta fragment is writer-friendly: it contains the remaining, smaller part of the data, uses only non-sorted dictionaries and N-bit compression.

For both the Main and the Delta fragments, snapshot isolation is implemented separately of the data, using dedicated MVCC structures. Queries access both the Delta and Main fragments. All new data is inserted into the Delta fragment, and the MVCC structures are modified accordingly. The deletion is logical; it affects only the MVCC structures. There are no in-place updates, they are implemented as insert plus delete.

When the Delta fragment becomes too large, it is merged into the Main fragment and a new empty Delta is created. The Delta Merge process is non-blocking to readers and writers, except during two short windows at the beginning and at the end.

Large tables are physically partitioned (hash, range, etc.). If a table is composed of n partitions, then each column has $2n$ Column Fragments, two for each partition, n Delta Column Fragments and n Main Column Fragments. If this table has m columns, then it has in all $2mn$ Column Fragments.

3. RELEVANT NVM FEATURES

We discuss here the NVRAM features relevant to its integration in HANA.

As discussed in the next section, we focus in this paper on the DIMM form-factor of NVM, that we call hereafter NVRAM, to stress the "random access" part of it. Although the SSD form-factor has itself a lot of merit, this is not the avenue that we have chosen in HANA. We discuss in section 4.1 the reasons of our choice.

In the DIMM form-factor, the next-generation of NVRAM ([7]) will have impressive qualities. It will provide performance that is close to what only DRAM can achieve today. Furthermore, it will exceed the capacity of DRAM devices at a lower price per GB. Attached to a CPU socket memory channel, its use is transparent (except for its persistency): it is directly accessible using load/store instructions, byte-addressable, and covered by CPU cache line handling, including cross-socket cache coherency. Applications can access it directly without additional copies in DRAM. Likewise, the NVRAM data transfers, similar to DRAM data transfers, are fine-grained at the size of a cache line instead of blocks. Last but not the least, NVRAM being non-volatile, its contents will not be lost on system restart.

However, there are several limitations of this new type of memory. The access latency is expected to be higher than that of DRAM, while the bandwidth is expected to be lower. An asymmetry is expected between reading and writing, i.e. writing will be slower than reading. Alas, beyond the qualitative characteristics listed above, the final quantitative characteristics of NVRAM DIMMs are not available at the point this paper was written. The experimental method used later in this paper copes with this by measuring with several latency scenarios.

4. NVRAM ADOPTION IN HANA

The problem at hand is how an in-memory database like HANA can take advantage of NVRAM while minimizing the negative impact of its drawbacks. We have focused on an early adoption, where HANA consumes NVRAM without heart surgery on the core relational engine. In this section, we discuss our choices, present the NVRAM adoption architecture, and list the challenges faced by the implementation of the chosen architecture.

4.1 Architecture choices

A first architectural question is whether to adopt the NVM technology as a faster persistent block device (i.e. using the SSD form-factor) or as a larger and persistent RAM (i.e. using the DIMM form-factor). Although the block access of the SSD form-factor might be well-suited for a traditional, disk-oriented DBMS, where the page buffer cache is the natural integration point, we took a different avenue in HANA. HANA is an in-memory database and does not handle disk blocks in a buffer cache. It operates on in-memory data and the Column Fragment is the unit of loading into and offloading from RAM. We have chosen thus to investigate an architecture where the in-memory Column Fragment is likewise the unit of placement in NVRAM, i.e. to use the DIMM form-factor NVM, which is part of the process' address space. We are leveraging thus both the promise of larger RAM capacity and the immediate availability of NVRAM data at system restart, where reading data from disk is not needed. Last but not least, HANA's memory-oriented algorithms simply work as such, without the need for substantial changes to obtain an NVRAM-enabled binary. Indeed, NVRAM is to the CPU another kind of RAM. We have likewise chosen to leverage both the

larger size of and the persistency of NVRAM, even if each of them is by itself interesting to HANA.

A further practical consideration comes from our scope: NVRAM early adoption for productive use, not only for prototyping, and not involving a full system redesign. HANA has grown into a mature DBMS, with datacenter-ready features, such as advanced backup and recovery, system replication, HA and DR support, etc. Many of these features are implemented based on the existing persistency layer: on-disk data and redo log volumes. For instance, one of the HA techniques supported by HANA is based on a standby node acquiring the disks of a failing node. Likewise, data backup accesses directly disk pages; etc. Re-implementing all these features based on NVRAM, as part of the early adoption, would have been a huge effort and would have diluted our focus. Discontinuing them would not be an option for productive use. We have thus decided to keep the disk-based persistency, and adopt NVRAM beside it. This gave us also the choice to place only a sub-set of the data in NVRAM.

Our next question was what to place into NVRAM. For this, we noticed that the Main Column Fragment data structures are natural candidates, for several reasons. From the perspective read accesses, sequential scans fare better in NVRAM than point reads: the cache line pre-fetch is expected to mitigate the higher latency. Within the Column Fragment data structures, the column vector data is sequentially scanned in many cases, more often within analytical workload. Conversely, point reads are more likely to produce cache line misses, thus exposing the higher latency of NVRAM. And point reads also occur within Column Fragment data, mainly in OLTP queries. However, an OLTP query touches few rows, and little column vector and dictionary data is accessed; most of the processing is done on intermediate data structures. We have decided to place in NVRAM only the very large Column Fragment data structures, i.e. the backing arrays of the column vector, dictionary, etc., and to keep the column store's smaller granularity in-memory objects and the query plan's intermediate results in DRAM.

As for the write access, the relevant factor is the integration of the NVRAM's different persistent behavior into HANA. Here, frequent NVRAM writes would be problematic. When only part of the column store data is persisted in NVRAM, HANA needs to explicitly keep in-sync, through some mechanism, the traditional HANA persistency layer and the NVRAM structures. Luckily, updates of the Main Column Fragment data are infrequent; actually the only writer is the Delta Merge process. Conversely, the write-frequent Delta Column Fragments and the MVCC structures, if placed in NVRAM, would exercise frequently this sync-up mechanism and thus expose its overhead. We have decided thus to place only Main Column Fragment data structures in NVRAM.

4.2 The NVRAM adoption architecture

To summarize, the early adoption architecture is thus for NVRAM to replace DRAM for the Main Column Fragment structures having the larger footprint. The large memory allocations of Main Column Fragments are placed directly in NVRAM, notably the dictionary and column vector backing arrays. Each Main Column Fragment has an associated NVRAM block containing them. As illustrated in Figure 5, smaller intermediate data structures of Main Column Fragments are still allocated in DRAM. Likewise, all Delta and MVCC data structures are allocated in DRAM. The intermediate results of query processing also continue to be allocated in DRAM.

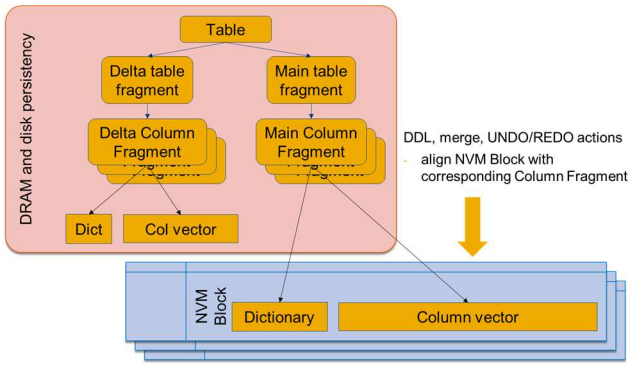

Figure 5: Usage of NVRAM for columnar data

When a Main Column Fragment is loaded into memory, its large data structures are not allocated in DRAM and initialized from disk anymore. Instead, each Main Column Fragment locates its associated NVRAM block and points directly to its column vector and dictionary backing arrays.

4.3 Challenges

However, implementing this architecture is not straight forward; a few challenges must be overcome.

A difficulty when placing data structures designed for DRAM into NVRAM, is the handling of the pointer values. Each time HANA starts, all the memory allocations (including NVRAM blocks) are mapped to different virtual memory addresses. Any pointer persisted in NVRAM therefore will need to be remapped. Our solution, discussed in section 5, is not to place any pointers in NVRAM. This solution is a natural fit since the large memory allocations are the leaves of the HANA column store data structures and hold no pointers.

The other difficulty is to integrate the NVRAM blocks, whose persistency has a different life cycle, into an existing transactional store which persists its in-memory contents to disk pages and implements ACID transactions based on checkpoints and REDO/UNDO logging. Our solution, discussed in section 6, is for the Main Column Fragments, whose lifecycle is driven by Delta merges and DDLs (as adding/dropping of column), to drive the lifecycle of their associated NVRAM blocks. Notably, they create and logically delete them as needed. In HANA, the checkpoint (which truncates the REDO log) will also physically destroy logically deleted NVRAM blocks: without this delayed destruction, replaying the REDO log might recreate a Main Column Fragment with dangling NVRAM references.

Finally, there are filesystem-specific challenges. The SNIA NVRAM Programming model extends the semantics of file systems for persistent memory [12]. The NVRAM technology on Linux will be based on DAX ("Direct Access") which avoids any filesystem caching [11]. Linux filesystems often create files with holes by default (also called "sparse" files). The advantage of this approach is obviously that storage is physically allocated only at the time of actual writes to the file. With the DAX-based architecture, NVRAM relies on user-space flushing (using hardware instructions, e.g., CLFLUSH, MFENCE, etc.), which may not work if HANA does a store to a location in a DAX file with a "hole", since the store causes a page fault so the file system can fill the hole with a new allocation. The file system metadata that gets updated by that fault may not be flushed to persistence until msync() is called. So, if we just relied on hardware primitives to flush the store, HANA could crash before the metadata changes are written and the stores would be lost, even

though the flushes had been performed. Hence, extra care will need to be taken to guarantee that filesystem metadata gets flushed.

4.4 Contributions of the architecture

The main architectural contribution of this paper is the specific placement in either NVRAM or DRAM of the different classes of data structures, for an in-memory DBMS running on a mix of both kinds of RAM. We discuss the reasons behind our approach, and highlight its advantages and challenges. In the next sections, we add specific contributions on how we have overcome some of these challenges.

Note that this architectural contribution is not limited to HANA, it is of a general nature. It also applies to other in-memory DBMS, such as Hyper [8].

5. COLUMN DATA FORMAT

We describe here the impact of NVRAM adoption to the in-memory columnar data format, as the changes done to the column vector and dictionary data layout.

5.1 Main column data persistence format

HANA Main column data consists of the dictionary, the column vector and optional additional data structures like indexes from dictionary value ids to the positions of the column vector using the id. Each of the column's data structures typically consists of some small descriptive data, the size of which is independent of the column size, and content data, which heavily depends on the column size. Examples for descriptive data are the number of rows or the number of distinct values of the column. Examples for content data are the content of the column vector, and the list of distinct column values in the dictionary.

Different implementations of the column's data structures are available depending on the column's data type or other table or column characteristics.

Since Main Column Fragments are only created during Delta merge, and are never changed after the merge, the Main Column Fragments only need to be written once and completely into persistence at the end of the merge.

HANA stores the Main column data in sequential files within an internal file system provided by the HANA persistence. To persist column data, the file is created and the column data is *serialized* into it. At load time, the file is *deserialized*, its content is copied to memory blocks allocated from DRAM.

Serialization writes the column data in a file format, which is closely related to the memory representation of the data. It consists of a format version number, the serialized dictionary, the serialized column vector, and optionally of the serialized additional data structures. Different implementations of the data structures define the specific format of the corresponding file section. They may also write a format version number specific for the corresponding section.

Typically format version numbers and the small amount of descriptive data are stored by value, while the large amount of content data is stored as a binary memory dump of the corresponding memory representation.

5.2 Format adjustments for NVRAM representation

With the availability of NVRAM storage we do not change the process of data creation for a new Main Column Fragment. The data is initially created in memory allocated from DRAM. This is because this data is frequently changed during the Delta merge till the data creation is finished.

When using NVRAM storage, we use the existing persistence format with only small adjustments to write the column data into the NVRAM block associated with the Main Column Fragment. This happens at the end of the merge before the new Main Column Fragment is accessible to other transactions.

The adjustments of the persistence format for writing into NVRAM consist of:

- Add alignment to memory blocks that need to be aligned in the virtual address space; conversely, the existing file serialized format has no alignment requirements.
- Prefer exact copies from memory rather than a format that allows simpler read on different platforms or is more efficiently compressed (see e.g. fixed-sized dictionaries below); conversely, the existing file serialized format is both multi-platform and further compressed.

The persistence format used for persisting the column data is not affected by these adjustments.

Before creating the NVRAM block, we do a "dry run" of the serialization to determine the required size of the NVRAM block. This is achieved by calling the serialization functions of the data structures with a special serializer instance, which just sums up the size of all data elements including alignments, but does not copy any data, and thus is extremely fast. This way we reduce the fragmentation and do not need to move the memory block to a different location when serialization ends.

During the write of a Column Fragment into NVRAM, only descriptive data and pointers to content data are kept in DRAM, the pointers are switched from DRAM addresses to the corresponding addresses within the mapped NVRAM block, and the previous DRAM memory used for content data is freed. When writing of the NVRAM block is finished, we keep the NVRAM block mapped in the virtual address space and access the data directly from the NVRAM block during all subsequent column accesses.

As a consequence of this design, we need to keep all of the NVRAM column mapped to memory, including the parts that we do not access after deserialization. This does not present a problem because the overhead is usually small and constant in size (not depending on the number of rows).

When loading a column, which had been written into NVRAM, we deserialize the column data from the mapped NVRAM block instead from the persistence file. We only copy descriptive data into DRAM, while pointers to content data are assigned to the corresponding addresses within the mapped NVRAM block.

We currently cannot update parts of a column without creating a new NVRAM block that contains copies of the data we do not need to update, e.g. the dictionary in case it is unaffected by the change. This is not a new restriction however, and we decided against optimizing in that direction before in the past, for reasons that still exist like it would require to manage more than one file per Column Fragment, adding overhead for that data and the files life cycle management.

5.3 Format adjustments for dictionaries

HANA stores the distinct values of a column in a dictionary. Dictionary data consists of a small amount of descriptive data, e.g. the number of values, and the list of distinct values.

Depending on the column's data type all column values may have the same byte length (numerical types, data/time types, etc.) or may have variable byte lengths (strings, binary data of variable length). Different dictionary implementations are used in HANA to handle these two cases.

5.3.1 Fixed-sized dictionaries

Fixed-sized dictionaries are used in case all column values have the same byte length. These dictionaries use an array implementation for the list of distinct column values. This array is the only content data of the dictionary.

In the persistence format the values traditionally are converted into compressed strings to be agnostic to data type, platform, or software changes and have small robust code at the same time.

Since access to any value in this format requires a re-conversion, the format was adjusted for writing into NVRAM blocks:

- Add alignment to the start address of the value array within the NVRAM block.
- Write the value array as memory dump into NVRAM.

During deserialization, a pointer to the value array residing in DRAM data is just assigned to the start address of the array within the mapped NVRAM block.

5.3.2 Variable-sized dictionaries

Variable-sized dictionaries are used in case the column values may have different byte lengths. These dictionaries use more complex implementations for the list of distinct column values.

Typically, the values are encoded as a sequence consisting of the number of bytes and the bytes themselves. But additionally, an index structure is used, which stores the start address of each n-th value, where n is some small constant, e.g. 16. This is needed to access values of a given index quickly. Only a limited memory range containing at most n values must be searched.

This additional index may contribute to a considerable percentage of the memory consumption of the whole dictionary in particular, if the values have small byte length on average. The additional index cannot be stored in NVRAM, if it consists of absolute memory addresses.

In order to also store this additional index in NVRAM, and to use it from the mapped NVRAM block after deserialization, we changed its internal format to use memory offsets instead of absolute memory addresses.

5.4 Contributions of the NVRAM column data format

The main contribution to column data format is the choice of a layout which allocates in NVRAM the large memory blocks, as the column vector and dictionary backing arrays. Also, our layout does not place in NVRAM any pointer.

As a further contribution, we have described in sufficient detail HANA's in-memory columnar data format.

Beyond the detailed HANA design, these principles have sufficient generality to be applied to other in-memory DBMS.

6. COLUMN STORE

We describe here the impact of NVRAM adoption to the column store, as the association of an NVRAM block to its corresponding Column Fragment and the driving of the NVRAM block lifecycle.

6.1 Lifecycle of NVRAM blocks

In HANA, Column Fragments are created and destroyed by the Delta merge (all Column Fragments of the relevant Table Fragments) and by DDL (all Column Fragments of the relevant columns).

Irrespectively of Delta merge or DDL, the creation of a new Main Column Fragment must trigger the creation of a new NVRAM block and the removal of a Main Column Fragment must trigger the removal of its NVRAM block. Transactional guarantees with regards to creation and removal of each Main Column Fragment must also be provided for the associated NVRAM block.

In HANA, a Main Column Fragment is represented by a persistent descriptor and a paired transient object. The persistent descriptor contains the Column Fragment's persisted state (i.e., the persisted column data and metadata), offers limited functionality, and refers to other metadata persistent descriptors through persistent pointers. The paired transient object points to the persistent column descriptor, enriches the persistent descriptor's functionality and state, and refers to other metadata transient object through handles.

To support persistence of data in NVRAM blocks, the persistent column descriptor has been extended with an NVRAM block id, a numeric identifier, used to determine the NVRAM block associated to the Column Fragment. NVRAM blocks in HANA are identified by a string key (see section 0). A Column Fragment NVRAM block key is constructed based on the table, fragment, and column identifiers and suffixed with the Column Fragment's NVRAM block id. On column load, the column transient object provides functionality to obtain its corresponding NVRAM block resulting in the mapping of the NVRAM block to a memory address in HANA's memory space.

The logical lifecycle of an NVRAM block is dependent on its Main Column Fragment. When a Main Column Fragment is created its NVRAM block is created as well. At this point in time this newly created NVRAM block is not marked as committed and thus it will be removed on system restart or failure by the operation (e.g., DDL or Delta merge) for which the Main Column Fragment is being created.

A consistent change (CCH) is a mechanism used in HANA to block the database checkpoint. The checkpoint cannot run while there is an active CCH. Within a single CCH an undo log record ('undoCreateNVMBlock') is written containing the NVRAM block, column, fragment, and table ids. Also within the same CCH the NVRAM block is NVRAM-committed by the NVRAM Block Provider and the NVRAM block id value in the persistent descriptor for the column is updated to the value used to construct the NVRAM block key for the NVRAM block. An NVRAM block will be NVRAM-committed at most once, and once committed it will never be modified again. From this point onwards, we must guarantee the NVRAM block will exists for the life of the Main Column Fragment or until it is replaced by a new NVRAM block if the data representation of the Column Fragment changes (more on this below).

HANA's NVRAM Block Provider (see Section 7) guarantees NVRAM-committed NVRAM blocks that exist at checkpoint time persist after a restart of HANA (i.e., each NVRAM-committed and checkpointed block exists and its contents are the same as at the time the NVRAM block was NVRAM-committed). Similarly blocks that have been created but not committed, and blocks that have been committed but not have been covered by a

checkpoint are physically removed by the HANA NVRAM Block Provider when recovering after a crash.

When a Main Column Fragment is removed (for example because the column is dropped) the associated NVRAM block is *logically* removed. The removal of the NVRAM block is done via a HANA callback mechanism that is invoked for each structure owned by the Main Column Fragment at the time the Column Fragment is about to be removed. The actual removal of the NVRAM block is logical: only until a checkpoint occurs NVRAM blocks that have been requested to be removed get physically removed. Main Column Fragments are removed at: cleanup of the transaction that has removed the column (e.g., alter table drop column); or, at cleanup of the transaction that has removed the Table Fragment where the Main Column Fragment is contained (e.g., removal of the old Main Table Fragment after a Delta-to-Main merge); or at undo, when the transaction that created the Column Fragment is rolling back.

Cleanup is an asynchronous process in HANA, in charge of physically removing persisted structures, both in disk and NVRAM. Operations (e.g., dropping of a column) schedule actions (e.g., physical removal of the Column Fragments) that must occur at cleanup. When it is guaranteed that there are no transactions in the system that can still view the affected object (e.g., the dropped column), the cleanup actions that were scheduled are executed. The HANA cleanup process therefore advances as transactions complete. The changes (both on disk and NVRAM) performed by cleanup are only persisted (i.e., made permanent) at checkpoint.

When an NVRAM block has been logically removed and the server crashes before the logical removal is checkpointed, HANA's NVRAM Block Provider guarantees the block is not removed during recovery and the logical delete operation is discarded.

NVRAM Blocks can also be deleted even if the owning Main Column Fragment is not removed. In other words, a Column Fragment may be associated to different NVRAM blocks throughout its life. An example of this situation in HANA is when the data of the Main Column Fragment is reorganized for better compression. In this case, no new Main Column Fragment is created: the current NVRAM block associated to the Column Fragment is deleted and a new NVRAM block is created. In this case: the Column Fragment is repopulated (its data representation changes); a new NVRAM block key string is constructed by using the current NVRAM block id for the Column Fragment plus one; a new NVRAM block is created and populated based on this NVRAM key; within a single CCH we write the 'undoCreateNvmBlock' as previously described. The old NVRAM block is scheduled to be deleted by HANA's cleanup mechanism. Within the same CCH, the NVRAM block is NVRAM-committed and the NVRAM block id in the Column Fragment descriptor is updated.

6.2 Recovery Considerations

From this point onwards it is guaranteed: if the transaction is committed, the old NVRAM block is deleted and the new NVRAM block will exist for the life of the Column Fragment or until it is replaced. The removal of the old NVRAM block is guaranteed by HANA's cleanup mechanism previously described. If the transaction rollbacks, the new NVRAM block is discarded, the old NVRAM block is re-established as the NVRAM block for the Column Fragment. The re-establishment of the old NVRAM block is guaranteed by HANA's undo mechanism.

We now illustrate error and crash recovery handling for the creation and population of the Main attribute NVRAM blocks and their association with their corresponding Column Fragment. The Main Column Fragment and its associated NVRAM block can be created independently. Only the writing of the 'undoCreateNvmBlock' undo, the NVRAM-commit and the association of NVRAM block to the Column Fragment need to occur within a single CCH. Let's assume a Main Column Fragment and what will be its NVRAM block have been created. The NVRAM block has been allocated but it has not yet been populated: if a crash occurs at this point, the HANA NVRAM Block Provider reclaims the NVRAM block as part of the recovery. Specifically, the HANA NVRAM Block Provider guarantees an NVRAM block that has not been NVRAM-committed is deallocated in case of a crash. If the operation that created the Main Column Fragment and its NVRAM block needs to abort, for example due to an exception: the NVRAM block is destroyed as part of the exception handling without ever been NVRAM-committed.

Within the CCH cleanup is scheduled such that the NVRAM block being replaced (if any) is removed after the operation that triggered the creation of the new NVRAM block commits. If we crash at this point, the undo log record was not checkpointed and HANA's NVRAM Block Provider will simply take care of reclaiming the NVRAM block. Because changes to the Column Fragment were not persisted by a checkpoint the Column Fragment will still be associated with the NVRAM block prior to the creation of the new NVRAM block. If the crash happens after the CCH and the actions executed within the CCH were checkpointed, the processing of the undo reverts the change to the Column Fragment's NVRAM block id and logically deletes the new NVRAM block. If the transaction that caused the creation of the Main Column Fragment and NVRAM block rollbacks, undo for the transaction is acted upon. Specifically undo reverts the change to the Column Fragment's NVRAM block id and logically deletes the new NVRAM block.

Logical deletion of NVRAM blocks is not done as part of the execution of an operation (e.g., drop column). This is because there might be old readers which may be reading the old NVRAM block. If the block is logically deleted, and the block deletion is checkpointed, the NVRAM block will be lost and the operation cannot rollback. Hence the logical deletion of Column Fragment NVRAM blocks is done at cleanup or undo of the operation.

If the creation and population of a Main Column Fragment is redone during recovery, the recreation of the NVRAM block is done as well with no further NVRAM-specific redo required.

6.3 Column store requirements for the NVRAM Block Provider

We now describe the NVRAM Block Provider's features that are critical at the column store level for providing database transactional properties for the lifecycle of NVRAM blocks.

NVRAM blocks that have not been NVRAM-committed do not persist after a crash. This guarantees allocated NVRAM blocks not yet committed are not leaked after a crash. NVRAM blocks that have been NVRAM-committed but the NVRAM-commit has not been covered by a checkpoint are deleted during the recovery after a crash. This guarantees we do not leak NVRAM blocks when a high-level operation (e.g., add column, merge) creates an NVRAM block but the undo of the operation is not persisted.

In HANA, the Delta merge is not durable unless all the modifications made by the merge and the commit of the merge transaction are checkpointed. The merge is a physical data reorganization, neutral to transactional visibility, is not redo logger, and not replayed by the recovery. Actions performed by a Delta merge are therefore lost after a crash if the commit of the merge was not followed by a checkpoint prior to the crash. When an NVRAM block has been logically deleted and this logical delete has not been checkpointed, and assuming the NVRAM-commit of the NVRAM block has been checkpointed, the NVRAM block persists after a crash and the request to delete the NVRAM block is discarded by HANA's NVRAM Block Provider. This guarantees Column Fragments contained in the re-established Main fragment after the undo of the merge will have valid NVRAM blocks.

Similarly, if the NVRAM-commit of a block has been checkpointed, then the NVRAM block persists after a crash. To keep or not to keep the NVRAM block will be decided during recovery based on whether the transaction is undone (i.e., if not to be kept, the logical removal will be triggered by the undo handling executed for the transaction). If a logical remove of an NVRAM block has been checkpointed, then the NVRAM block does not persist after a crash. This a requirement because, due to the checkpoint, there is no way to redo the logical remove.

6.4 Contributions of the column store integration

The main contribution of the store integration is the alignment of the NVRAM block lifecycle with the lifecycle of column store persistent artefacts, which is itself driven by Delta merge and DDL.

As a further contribution, we describe the relevant HANA column store durability mechanisms, as the CCH, checkpointing, undo/redo logging, transaction cleanup, etc.

Beyond the HANA specifics, the solution can be applied to any DBMS whose persistence is based on write-ahead redo logging and checkpointing dirty data pages.

7. PERSISTENCY LAYER

We describe here the impact of NVRAM adoption on the persistency layer, which provides the underlying mechanisms of handling NVRAM blocks.

7.1 Need for consistent management of data

In some situations, the operating system may expose persistent memory as a memory-mapped file, but an application would not want simply such a raw form of access. As the memory is now persistent, we not only need to have heap functions like malloc/free but also a way of keeping the data consistent across system failures. For example, if a process/program terminates due to an error after a persistent memory allocation call, then this will leave a persistent memory leak which is more dangerous than in case of volatile memory, where the termination itself cleans all such leaks. So, in order to have a correctly and efficiently operational DBMS which uses NVRAM, it is important to have a sound and consistent management of the physical memory regions on the NVRAM DIMMs. As outlined in the next sections, our solution is to provide interfaces and building blocks to exploit the underlying hardware primitives for flushing the data and guaranteeing consistent and durable write operations.

7.2 Introduction to NVRAM Block Provider

The *NVRAM Block Provider* of HANA is a module consistent persistent memory management so that the upper layers of DBMS can seamlessly exploit the persistent memory space without having to worry about factors like allocation, deallocation, data durability, persistent memory leaks, and recovery of data after a restart. The NVRAM Block Provider library sits between the OS/hardware and the upper layers of HANA, and uses a directory on a mounted filesystem, where NVRAM blocks are stored as files and mapped into memory. Currently, the following storage types are supported:

1. DAX-enabled filesystem backed by real NVRAM DIMMs
2. Traditional file system backed by SHM/SSD/HDDs (Simulation)

The programming model is based upon memory-mapped files, and hence the NVRAM Block Provider not only leverages the load/store capabilities of persistent memory, but can also work correctly with any other storage technology (e.g. SSDs).

7.3 Design of the NVRAM Block Provider

The implementation model of NVRAM Block Provider stores NVRAM blocks as files into a root directory of the mounted filesystems. When working with real hardware, this should be the location where one has mounted the persistent memory devices (or DIMMs). We expect the physical location to be DAX-enabled, which is a Linux feature added for persistent memory storage [11]. If supported, the DAX bypasses the kernel page cache which is usually used to buffer reads/writes into files. For file-based mappings, the persistent memory space is mapped directly into user space. For simulated mode of testing, the root directory could be any path under your file system. We store all the NVRAM-resident blocks under this root directory and at startup load all the blocks from the same fixed directory. As we are dealing with memory management which is persistent, the NVRAM physical block lifecycle management is kept consistent with HANA's checkpoint logic of data persistency and is thus driven by a checkpoint/restart/crash at any point in time. For supporting database backups and recoverability, we keep another copy of the data in a standard persistence on disk, and also log to normal log files on the disk. This design may be optimized in the future. NVRAM Block Provider keeps track of the checkpoint version of the system at the time of creation of a new NVRAM block and uses the information to handle blocks during every other event (i.e. checkpoint/restart/crash) which helps to immensely simplify the work of the upper layers (referred as 'client modules') of HANA. For every NVRAM block allocation request, NVRAM Block Provider takes as input a key (or name) that uniquely identifies a block along with the requested size of allocation. The block management module maintains the following types of blocks based upon the three possible lifecycle states of a block (state transitions are illustrated in Figure 6):

Data block: Stores the actual data of relational tables. This is a closed (a.k.a. NVM-committed) and persisted block (in fact marked as "read-only"), flushed using the processor's persistency instructions/primitives provided by the hardware vendor.

Tombstone block: Stores metadata for transient delete operations. This is created when a data block that belongs to an earlier checkpoint version is requested for deletion. This block is created to persist the information that a block is marked for deletion so that we can handle such blocks even in case of system crash before actual deletion.

Temporary block: An uncommitted data block, that was created but the system crashed before it could be written and flushed (i.e. NVM-committed) to persistent memory space. The information is persisted in the block's name so that it can be differentiated from the valid set of blocks that survive after a crash. This is generally pruned at checkpoint/restart time.

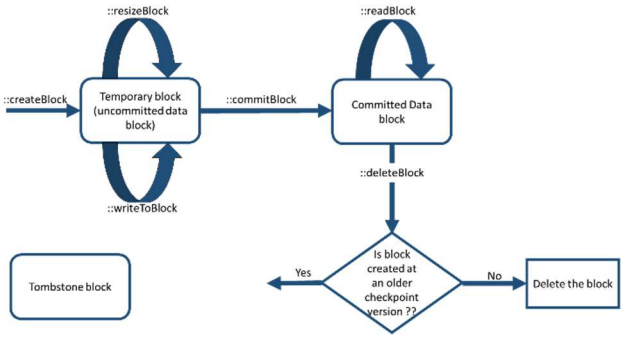

Figure 6: State transition diagram of NVRAM blocks

To ensure efficient access and cleanup of these blocks, the NVRAM Block Provider maintains three key data structures in DRAM:

Data Block Map: A sorted multi-map of the NVRAM data blocks to ensure easy access to the blocks. On every successful creation of a NVRAM block, an entry is added into the map. Each block entry maintains the vital information regarding a data block state ('NVM-committed' or 'marked for deletion'), file size, a handle to its virtual address mapping and path location.

Tombstone Block Map: A map of tombstones. A checkpoint ensures that the system's state is consistent till that timestamp. If blocks belonging to earlier checkpoint versions are requested for deletion, an entry for such blocks is added into this map. The cleanup of such blocks happens as part of the next checkpoint. This map essentially helps in faster lookups of such blocks. If the system crashes before such a cleanup can be performed, this map is re-created upon startup using the earlier mentioned tombstone blocks persisted on NVRAM.

Temporary Block Map: A list of uncommitted and unhandled blocks. The list is processed upon server restart during checkpoints to clean-up these transient blocks.

We next explain the various APIs provided by NVRAM Block Provider and how they help to drive the lifecycle of a block. This is illustrated through the flow diagram in Figure 7 depicting a few possible scenarios. Upon every system startup, due to normal restart or crash, the following operations will take place:

1. NVRAM Block Provider is initialized with a root directory.
2. The root directory is traversed for persistent memory (NVRAM) blocks which are classified into the 3 categories mentioned before, and the three data structures (*Data Block Map, Tombstone Block Map, Temporary Block Map*) are subsequently populated.
3. A valid and consistent checkpoint based upon HANA's persistency state is fixed and installed globally.
4. All temporary blocks are cleaned up which helps to avoid 'persistent leak' situations as illustrated in Figure 8
5. Additionally, based upon the current checkpoint version:
   - All data blocks created at a later (or higher) checkpoint version are deleted
   - All tombstones created at a later (or higher) checkpoint version are deleted and the corresponding data blocks are marked as valid and NVM-committed
   - All tombstones created at an earlier (or lower) checkpoint version are deleted along with the corresponding data blocks

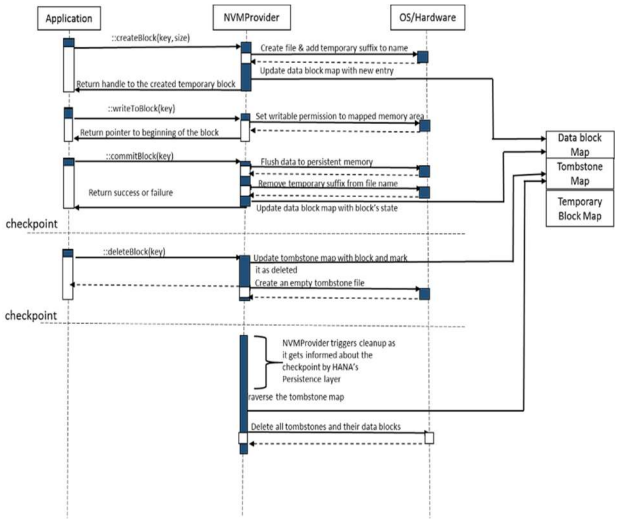

Figure 7: Life cycle management of NVRAM blocks

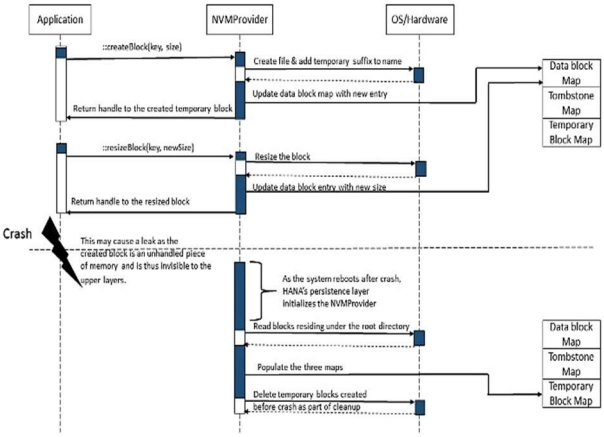

Figure 8: NVRAM Provider avoids persistent memory leaks

7.4 Contributions of the NVRAM block management

Our block handling scheme provides a consistent management of persistent memory space that appreciates the functioning of an In-Memory Data Management system such as SAP HANA. It maintains lookup maps for fast and efficient block searches of committed, uncommitted and defer-deleted blocks. We follow a unique and simple approach to persist a blocks' state information, necessary for recovery and to handle persistent memory leaks. The underlying directory structure maintained by NVRAM Block Provider is distinctive in a way that helps to retain/destroy blocks based upon their checkpoint versions. The NVRAM block management layer is responsible for management of a block's physical state across system checkpoints/restarts/crashes.

Furthermore, the NVRAM Block Provider is designed keeping in mind the requirements of HANA, but is completely independent

of its storage architecture i.e. Row-based or Column-based. The APIs allow clients to create arbitrary sized persistent memory blocks with resizable characteristics. We maintain statistical information like count of successfully created blocks, deleted blocks, loaded blocks, pruned blocks, etc. along with total size per block.

8. EXPERIMENTAL DATA

For evaluating the performance of our implementation before hardware availability, we are using the same custom setup as in [6]. This special processor and BIOS has the capability to add memory latency to certain memory regions. The system is equipped with 2 Intel Xeon processors E5-4620 v2 with 8 cores each, running at 2.6 GHz without Hyper-Threading. Two of the four memory channels are treated as "persistent memory" where the additional latency is applied. The remaining memory is untouched and can be used as DRAM. Our system was equipped with 256GB of memory, with 128GB configured as PMEM and 128GB as DRAM

8.1 Evaluation of runtime impact on OLTP and OLAP workloads

We first evaluated the impact of additional persistent memory latency on OLTP. The first experiments consist of random inserts in a table with 4M rows and 500 columns. All data in the table resides in Main store, which was allocated in persistent memory. The total running time 100,000 inserts is shown in Figure 9. As, by design of HANA, inserted data is stored in the Delta store, increasing the latency for persistent memory does not have a significant impact on performance. This can be seen as a best-case scenario in respect to memory latency for the Main store.

We then evaluated "single selects" in Main store, which is probably the worst-case scenario in this respect. The same table was merged and kept in Main store during the test. Figure 10 shows the running time for different persistent memory latencies. In case of the column orientation, the individual cells of a row reside in different memory locations. Reading a complete row therefore reads (at least) one cache line for each cell. Since the table does not fit in the processor cache and the lines are read randomly, this results in at least one cache line miss per cell, which is fully impacted by the latency of persistent memory.

In order to evaluate the impact of higher latencies on OLAP workloads, we compare the throughput for a variation of the TPC-H workload. The tables in the benchmark have been widened to better reflect the tables sizes observed in typical SAP systems [5]. Furthermore, the queries were modified to better mimic the interactive data exploration, which is common in today's in-memory solutions. During the tests, all tables are allocated in NVRAM, whereas DRAM is used for the remaining data structures. Figure 11 shows the average throughput depending on the latency of the NVRAM region. The leftmost data point reflects the measurement with not additional latency applied, i. e. the DRAM latency for this setup. The impact on throughput with increasing latency is very small, as the access pattern for OLAP workloads is very predictable and the hardware prefetchers can easily hide the additional latency.

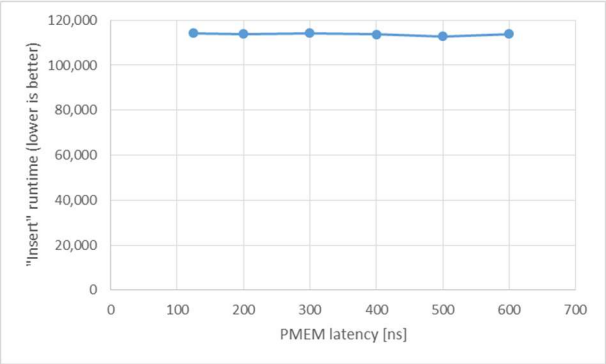

Figure 9: "insert" performance

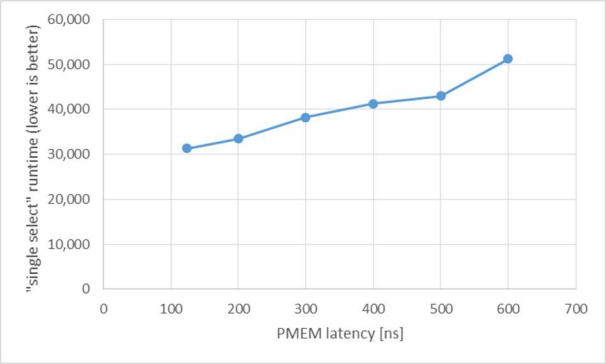

Figure 10: Worst case scenario of "single selects"

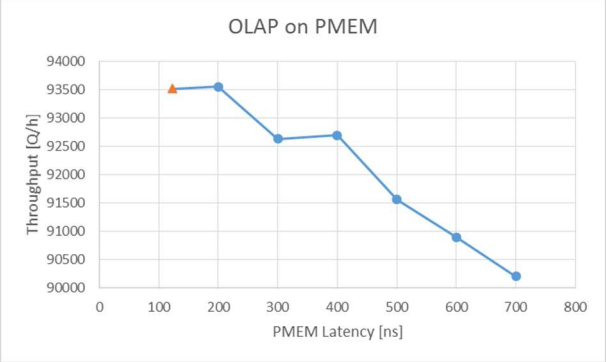

Figure 11: Throughput vs. latency for OLAP

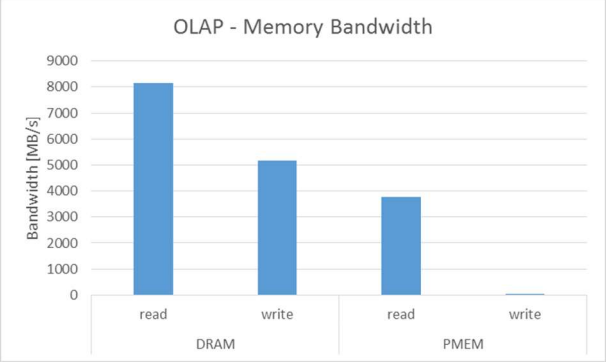

Figure 12: Bandwidth requirements for DRAM and NVRAM

Figure 12 depicts the memory bandwidth that is used during the benchmark run for the different memory tiers. As the Main is not updated during the benchmark run, there is only read traffic from the persistent memory region. Furthermore, by the nature of how

HANA processes data in aggregation, there are many more reads and writes to the DRAM region. In particular, the bandwidth requirements for persistent memory are significantly lower than for DRAM.

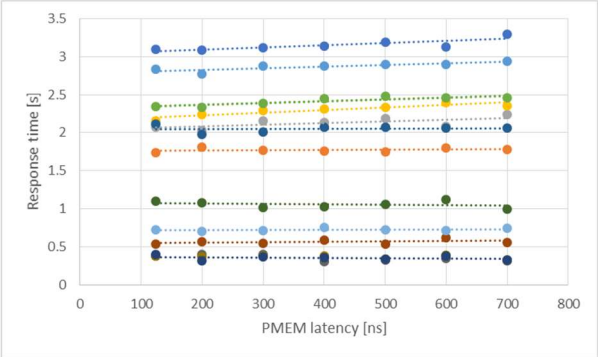

Figure 13: Response time per OLAP query

The impact of additional latency for individual queries is shown in Figure 13. Each data point is the average response time for a complete benchmark run. Again, the latency for the persistent memory region was varied, whereas data structures in DRAM region are not affected. One can observe that some of the queries are more sensitive to the latency of the persistent memory than others. This can be explained by multiple factors:

1. Does the query exhibit a memory access pattern that can easily prefetch by the hardware prefetchers?
2. Is the working set of queries small enough to fit in CPU cache and hence agnostic to persistent memory latency?
3. Is processing of the query compute or latency bound?

8.2 Evaluation of DBMS restart times and Memory footprint

The set of experiments mentioned below are focused on evaluating the potential benefits in terms of Database availability and TCO provided by HANA's architectural adoption of NVRAM. As mentioned in earlier sections, most of the large memory intensive data structures will now be stored on NVRAM. These structures are mapped into memory from NVRAM-based filesystems upon first access of a table, instead of copying them into DRAM from disk as was done earlier. This approach is expected to benefit HANA in two unique ways. Firstly, after a DBMS server crash or restart, the data can be made available near instantly. Secondly, as most of the heavy data structures are resident in NVRAM, the allocations in DRAM would reduce substantially. This should help to lower the overall system cost since customers can operate on sizable business data with a much smaller DRAM requirement. The system we use for conducting the experiment is Intel(R) Xeon(R) CPU E7-8880 v2 @ 2.50GHz with 1TB Main Memory capacity. We use a columnar table of size of approximately 5 GB with 4 million records and 100 columns that span different flavor of data types (e.g. integer, decimal and strings). The strategy is to use mmap-based filesystem simulation on Linux Shared Memory (i.e. /dev/shm) in the absence of real NVRAM hardware. We believe the "SHM"-based simulation approach is closest to the execution behavior of NVRAM-optimized HANA on real persistent memory DIMMs. The DRAM-based HANA is persisted on fast SSDs (Solid-State Drives) providing an aggregate read bandwidth of ~1200 MB/s. The strategy for NVRAM-based HANA is to use mmap-based filesystem simulation on Linux Shared Memory (i.e. /dev/shm) in the absence of real NVRAM hardware. We believe the "SHM"-based simulation approach is closest to the execution behavior of NVRAM-optimized HANA on real persistent memory DIMMs.

The Figure 14 illustrates the experiment where we vary the table size from 400 rows all the way to 4 million rows (i.e. 10,000x) in steps of 10x and calculate the time to "preload" the table immediately after a server restart. For in-memory databases like HANA, the "preload" operation is responsible for bringing the entire table into system memory thereby making it ready for online operations like queries. The figure shows that for increasing table size, the NVRAM-based HANA approach shows a flat trend with preload time being nearly constant. On the other hand, the preload speed with traditional approach based on DRAM is linearly dependent on the size of the table.

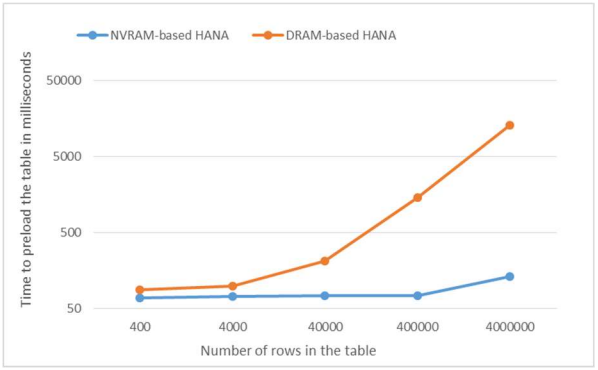

Figure 14: Table preload cost after server restart

We further perform an experiment to measure the TCO benefits of NVRAM storage by studying the change in DRAM consumption with increasing data populated in the target table. For this purpose, we vary the tabular data from 400 rows to 4 million rows and measure the DRAM consumption after table is pre-loaded as explained earlier. As seen from Figure 15, the results indicate that in case of NVRAM, the graph is nearly flat with a nominal increase in memory consumption with increasing data scales, unlike the DRAM case where all data structures are copied into Main Memory upon pre-loading of table.

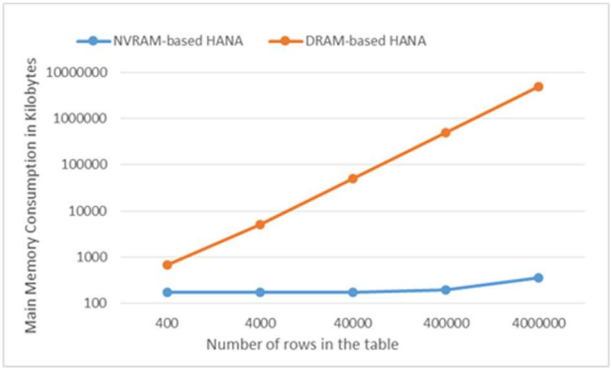

Figure 15: DRAM consumption upon data loading

The above set of experiments were additionally verified using several diagnostic tools provided by HANA. For the NVRAM-aware HANA case, we could see ~5GB being allocated from NVRAM based on Shared Memory (i.e. /dev/shm) when all 4 million rows of the table are loaded. Also, using CPU profilers, we observed that in DRAM case, majority of CPU cost is being spent in disk-based reads whereas for NVRAM, the disk reads are absent since we map data directly into process address space.

8.3 Contributions of the experimental data

Below are the salient takeaways based on the various experiments conducted with our NVRAM-based HANA approach:

- The "insert" performance of HANA is totally unaffected by simulation-based latency costs of NVRAM
- The simulation-based NVRAM latency cost also has a marginal impact on OLAP throughput
- The table "preload" operation with NVRAM-based HANA is tremendously faster than DRAM-based HANA. It shows a flat trend in response time and the gains are seen to accelerate with increasing growth of tables
- The NVRAM-based HANA shows a flat trend in memory consumption immediately after table preload with increasing growth of tables

The previous two points suggest that HANA can leverage NVRAM to go online and be available for business much faster especially with a negligible Main Memory footprint

9. RELATED WORK

At our best knowledge, there is no prior publication describing the adoption of NVRAM by a productive DBMS.

Many recent papers focus on specific data structures, specific system modules, experiments, prototypes, architectural proposals, etc. See for instance [13], [15], [16], [17], [9], [2].

Testing software based on NVM is a novel domain. Although relevant to the HANA adoption of NVRAM, we do not cover it in this paper. Such a testing framework is described in [14].

In terms of physical block management for the NVM space, the only other persistent memory block manager at our best knowledge is NVML [6]. NVML is shipped as a 3rd party library (named 'libpmem') which provides basic helper APIs to manage and use NVM/NVRAM. The functionality includes mapping the memory chunks from the NVRAM space, flushing the memory stores, and other assisting toolbox libraries:

a. maintain a transactional object store ('libpmemobj').
b. cache fixed-size objects in pmem ('libpmemblk')
c. do writes to a persistent log file ('libpmemlog')
d. use NVRAM as a pool of volatile memory ('libvpmem')

Although NVML covers a lot of functionality, it does not cover HANA's requirements:

- support for variable-sized block creation ('libpmemblk' cannot be used).
- support for very big blocks ('libpmemobj' cannot be used).
- exploit NVM/NVRAM not only as an extension to DRAM, but also for its persistency ('libvpmem' cannot be used).

It may be noted that block-based structures have their own overheads in terms of increased fragmentation of space and higher random access costs due to page-granular operations. HANA tries to avoid these pitfalls.

10. CONCLUSION

In this paper, we describe the direction and technological challenges of the NVRAM early adoption within the SAP HANA DBMS. We have found the original HANA architecture and data structures well suited for NVRAM. Heart surgery is not needed for a very effective early NVRAM adoption. The main reason is that HANA is a new generation database, designed from scratch for in-memory access, and optimized for CPU cache efficiency, which is the main optimization for both DRAM and NVRAM.

This early adoption paper is however only scratching the surface, the problem space is enormous: placing other structures in NVRAM (Delta, MVCC, REDO/UNDO …), NVRAM vs. DRAM placement policies, etc. At the same time, we are investigating this larger solution space of the more radical integration of NVRAM (see for instance [13], [15], [16]).

11. REFERENCES

[1] Apalkov, D. et al. Spin-transfer torque magnetic random access memory (stt-mram). ACM J. Emerg. Technol. Comput. Syst., 9(2), 2013.

[2] Arulraj, J., Pavlo, A., and Dulloor, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In SIGMOD (2015), ACM, pp. 707-722.

[3] Burr, G. W. et al. Phase change memory technology. Journal of Vacuum Science & Technology B, 28(2), 2010.

[4] Dulloor, S. et al. System software for persistent memory. In EuroSys (2014).

[5] Färber, Franz et al.: The SAP HANA Database - An Architecture Overview. *IEEE Data Engineering Bulletin, Volume 35, Number 1, March 2012.* 28-33.

[6] Intel's NVML library http://pmem.io/nvml/

[7] Intel and Micron Produce Breakthrough Memory Technology (3D XPoint). https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[8] Kemper, Alfons and Neumann, Thomas. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE 2011: 195-206

[9] Kimura, H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In SIGMOD (2015), ACM, pp. 691-706.

[10] Krueger, Jens et al. 2011. Fast updates on read-optimized databases using multi-core CPUs. *Proc. VLDB Endow.* 5, 1 (September 2011), 61-72.

[11] Linux DAX https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[12] Number of nines availability of systems http://tanejagroup.com/files/Compellent_TG_Opinion_5_Nines_Sept_20121.pdf

[13] Oukid, Ismail et al.: Instant Recovery for Main Memory Databases. CIDR 2015

[14] Oukid, Ismail et al.: On testing persistent-memory-based software. DaMoN 2016: 5:1-5:7

[15] Oukid, Ismail et al.: FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. SIGMOD Conference 2016: 371-386

[16] Oukid, Ismail and Lehner, Wolfgang: Towards a Single-Level Database Architecture on Non-Volatile Main Memory. *To be published*

[17] Pelley, S. et al. Storage management in the NVRAM era. PVLDB 7, 2 (2013), 121-132.

[18] SNIA NVM Programming Model V1.1. Technical report, 2015.http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.1.pdf.

[19] Yang, J. J. and Williams, R. S. Memristive devices in computing system: Promises and challenges. ACM J. Emerg. Technol. Comput. Syst., 9(2), 2013.