Native Store Extension for SAP HANA

Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, Prasanta Ghosh
SAP SE
firstname.lastname@sap.com

ABSTRACT

We present an overview of SAP HANA's Native Store Extension (NSE). This extension substantially increases database capacity, allowing to scale far beyond available system memory. NSE is based on a hybrid in-memory and paged column store architecture composed from data access primitives. These primitives enable the processing of hybrid columns using the same algorithms optimized for traditional HANA's in-memory columns. Using only three key primitives, we fabricated byte-compatible counterparts for complex memory resident data structures (e.g. dictionary and hash-index), compressed schemes (e.g. sparse and run-length encoding), and exotic data types (e.g. geo-spatial). We developed a new buffer cache which optimizes the management of paged resources by smart strategies sensitive to page type and access patterns. The buffer cache integrates with HANA's new execution engine that issues pipelined prefetch requests to improve disk access patterns. A novel load unit configuration, along with a unified persistence format, allows the hybrid column store to dynamically switch between in-memory and paged data access to balance performance and storage economy according to application demands while reducing Total Cost of Ownership (TCO). A new partitioning scheme supports load unit specification at table, partition, and column level. Finally, a new advisor recommends optimal load unit configurations. Our experiments illustrate the performance and memory footprint improvements on typical customer scenarios.

PVLDB Reference Format:
Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, and Prasanta Ghosh. Native Store Extension for SAP HANA. *PVLDB*, 12(12): 2047-2058, 2019.
DOI: https://doi.org/10.14778/3352063.3352123

1. INTRODUCTION

The SAP HANA database (referred to simply as HANA) created a hybrid transactional and analytical processing paradigm, known as translytics or HTAP, by performing efficiently over the same shared data for both workloads. Performance for such combined workloads is achieved by combining the columnar store engine with highly optimized in-memory processing technologies. The combination of columnar store and in-memory processing optimizes performance by providing fast columnar access while avoiding per-column page access. HANA's original qualities were thus HTAP, multi-model, and performance. From the beginning, HANA's approach was to focus on qualities and treat features as being subservient to these desired qualities. Likewise, in cloud processing, the qualities of a service are more important than its features; a service with a high TCO or a low availability will fail to be adopted even if the service is feature-rich. HANA naturally evolves and extends its focus to cloud qualities, e.g. TCO, availability, elasticity, and automation.

HANA's Native Store Extension (NSE) is part of this major shift and one of the technology enhancements which reshapes HANA as a cloud-native database. The focus of NSE is to give HANA a cloud-native Infrastructure-as-a-Service cost structure with low TCO and the capability to cover a wide range of cost and performance choices while preserving HANA's original qualities. NSE extends HANA with a block device oriented store without introducing a specialized disk engine. Instead, we have taken a hybrid approach: the seamless extension of the in-memory store and engine. The extension is based on having a unified persistence format for in-memory processing and for paged access. This unified format is friendly to paging, enabling thus efficient disk processing, e.g. using a buffer cache, touching a minimal number of pages for point access and prefetching pages when appropriate for serial scans. The content of persistent pages is byte-compatible with contiguous sections of HANA's original in-memory columnar format. Therefore, the disk extension brings no performance degradation.

We extended the in-memory engine at the store access level and at the higher relational operator level. At the store access level (the leaves of the query plan), disk processing reuses the in-memory code within the limited scope of each page. The relational operators of HANA's execution engine are pipelined whenever on-the-fly processing is possible (e.g. for filter, project, hash probe). There are pipeline breakers whenever the whole input must be consumed before generating results. The execution engine's disk extension required no change for the pipelined operators. Only the pipeline breakers are extended with disk-based variants to limit their processing memory footprint, keeping HANA's in-memory processing performance the same for loaded columns.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352123

2. AN OVERVIEW OF SAP HANA

HANA supports both analytical and transactional access patterns [26, 20] and different data representations (structured, semi-structured and unstructured). HANA makes use of modern hardware features like multi-core CPUs [31], non-volatile memory (NVM) [6] and large main memory sizes. With parallelization at all levels (scale-up, scale-out) and MVCC-based transaction management [16], HANA provides scalability for increasingly complex enterprise workloads.

2.1 In-Memory Column Store

The relational capabilities of HANA are built around an in-memory column store. Columns are internally split into a read-optimized main fragment, handling the majority of the column data, and a small write-optimized delta fragment. Inserted or updated records are always appended to the end of the delta fragment. Queries are executed independently on both fragments and the two result sets are joined and returned, with some rows removed after applying row visibility rules. The tuples of the delta fragment are regularly merged into the main fragment by the delta merge operation. Data is stored using domain coding, specifically the column values are substituted by bit-packed value ids which point into a dictionary that contains the distinct, sorted (main) or unsorted (delta) original column values. Reverse lookups are accelerated by an optional inverted index. Depending on the data distribution, the value identifier array in main fragments is subject to further compaction using prefix, sparse, run-length, indirect, and cluster compression [18]. Domain coding, in conjunction with advanced compression, reduces memory footprint by up to a factor of 10x. The processing of queries in HANA is rapidly converging towards HEX (HANA Execution Engine), a new state-of-the-art query engine. HEX achieves great locality for queries using two features. First, HEX is able to merge adjacent physical plan operators and compile them into native code using LLVM. This is useful for CPU-intensive parts of a query where as much data as possible shall be retained in CPU registers. Second, HEX pipes tuple batches of approximately Lx cache size through a chain of plan operators. Compared to the traditional row-at-a-time model, pipelining amortizes operator invocation costs and improves cache behavior significantly.

2.2 In-Memory and On-Disk Processing

HANA's in-memory store serves many use cases well. However, database capacity is inherently constrained by the available amount of memory. As a remedy, SAP recently added NVM as a new storage option to HANA [6]. NVM continues to be a strategic direction, however, its size is expected to be only factors larger than DRAM whereas disk storage is orders of magnitude larger than DRAM. To substantially increase HANA's capacity and scale with the best cost/performance ratio, we extended HANA with native disk store technologies. Our solution implements a hybrid column store architecture which provides in-memory and page loadability across all compression types, data types, and database objects (schema, table, and column). With this new hybrid column store, HANA does not only enable seamless migration of data to slower storage as data volume increases, it also retains full functionality with predictable and elastic degradation of performance in resource-constrained environments such as cloud deployments.

3. ARCHITECTURAL OVERVIEW OF HYBRID COLUMN STORE IN SAP HANA

HANA's NSE adds complementary disk-based column store features to HANA's in-memory column store, which together form a flexible, hybrid column store. This hybrid column store offers in-memory processing for performance critical operations and buffer-managed paged processing for less critical, economical data access. This hybrid capability extends up from a unified persistence format which can be used to load paged or in-memory primitive structures, which in turn form paged or in-memory column store structures (data vectors, dictionaries, and indexes) that are ultimately arranged by the hybrid column store according to each column's load configuration. Hybrid columns can be configured to load all in-memory structures, all paged structures, or a mixture of both. This configuration can be manually set by the user or application, as well as automatically set based on workload, data access patterns, and intelligent algorithms. HANA's hybrid column store architecture is optimized towards in-memory processing in that the original HANA in-memory column store performance is preserved for in-memory configurations, while paged configurations are aligned as closely as possible to the in-memory structure to provide maximum code sharing at the top level. Fortunately, the existing set of in-memory column implementations (uncompressed and compressed simple values, as well as specialized representations for materialized hash index, persistent row id, and spatial data type) share some common high-level structures (dictionaries, indexes, data vectors and block vectors). These pieces can be encapsulated and interchanged with paged implementations.

The hybrid column store (Figure 1) is built on the concept of primitive building blocks. A primitive hides the memory or paged nature of those parts behind standard APIs. As an example, the most widely used in-memory primitive in the HANA column store is the n-bit compressed data vector, which provides a compressed representation of dictionary-encoded values in contiguous memory, while the paged counterpart provides the same compressed byte compatible representation in paged memory. Both primitives provide the same API but different memory footprint and performance characteristics. HANA's Data Definition Language SQL has been extended to allow the specification of load unit hints for database objects. A load unit hint declares the preferred access type (either paged or fully in-memory) of a database object. Load unit specifications allow for flexible multi-level table partitioning options, where all or selected table partitions can be declared with a given load unit tag such that the columns on the main fragments of the specified partitions follow a given load unit. New DDL statements are also provided for the efficient conversion of legacy database objects (originally created without NSE's unified persistence format) into the unified persistence format. An elastic buffer cache is provided to manage paged memory efficiently within a configurable size limit of the total memory allocated to the HANA database server. This limit is dynamically adjustable, and it works in tandem with the memory resource manager to ensure resources using paged memory do not interfere with memory resources allocated for in-memory objects.

An NSE Load Unit Advisor utility is provided to assist users with configuring the load unit for the database objects accessed by a given workload. The Load Unit Advisor

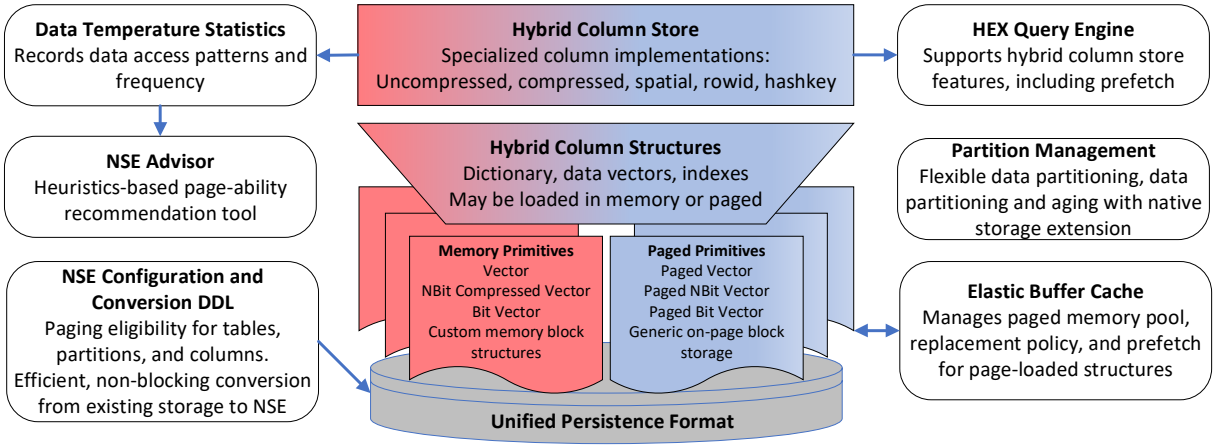

Figure 1: HANA's Hybrid Column Store Architecture

provides a heuristics-based engine that interprets data usage patterns collected by the hybrid column store to make recommendations on load unit changes; for example, that a table or column should be page loadable. These recommendations are aimed at placing smaller and frequently accessed database objects fully in memory while allowing larger and less frequently accessed objects in paged memory, balancing memory footprint with performance. Our contributions are presented as follows:

- We introduce the unified persistency (Section 4) which is consumed by the pageable primitives (Section 5), designed to integrate well with in-memory column store.
- We present the design of our new buffer cache (Section 6) and its integration with new execution engine (Section 7).
- We present metadata and online dynamic load unit conversion at column, table, and partition granularities (Section 8).
- We present the design of recommendation engine that provides hints for load unit conversion (Section 9).
- We share experimental results (Section 10) demonstrating memory footprint reduction and performance impact in end-to-end evaluation on large workload in difference scenarios.

4. UNIFIED PERSISTENCY FORMAT

Piecewise columnar access for read-optimized main fragments was previously introduced to HANA [29]. This paged access was designed to achieve minimal memory utilization as a complement to the high-performance data access algorithms developed and optimized for fully memory resident columns. This paged access empowered several use cases that relied on clear resolution of the load unit in advance. The configuration of load unit had to be done explicitly by the application expert who knows the data access patterns and knows the tradeoff between the memory consumption and performance. One of the use cases was data aging, where recently added data is considered hot, and configured to use in-memory load unit, while older data is considered warm and is configured to use paged load unit. When data is aged, hot data is converted to warm data by a lengthy persistence change from memory resident to page loadable. Once a column is chosen to be in either format, it will remain so throughout its life cycle. An explicit load unit conversion can change a column's persistency format (Section 8.2). However, the format conversion is slow, as it changes the format for every subcomponent of a column, i.e. the data vector, the dictionary, and any index present. With the introduction of a common persistence format, referred to as unified persistence in this paper, we bridge the gap between the two representations and avoid expensive format conversions when the load unit changes.

Unified persistence is achieved by providing an identical byte-compatible representation for each columnar data structure. This is achieved by composing primitives to construct a page loadable equivalent of any data structure of each column. If a subcomponent of a column needs to be fully loaded in memory for optimal performance, instead of needing a persistency format change, we switch the access mode for each data structure and primitive by allocating contiguous memory which is populated quickly and efficiently by copying data from temporary loaded pages (in the buffer cache) to memory. This step can be performed independently for each of the used primitives. When a subcomponent is small enough, e.g. less than the minimum page size, it can be co-located with other such subcomponents of the same column on the same page chain and it will always be loaded in memory directly and independently.

The unified persistency enables a new class of use cases by reducing the total cost of ownership for both in cloud and on-premise installations. Not only there is no need to perform a conversion between the two representations, but also we can dynamically load a column in either load unit type and perform this for each of its subcomponents. If needed, the applications can still provide a load unit type hint at table, partition or column level. However, this is entirely optional as the NSE Load Unit Advisor (Section 9) can analyze the workload of the application and provide the load unit type hints. Moreover, to achieve an optimal cost/performance ratio, a strategy can keep the most frequent used substructures of selected columns in memory and the others in paged format residing in a less performant media. This is mainly because the cost of switching from one load unit type to another is minimal (there is no persistency format conversion to switch load unit).

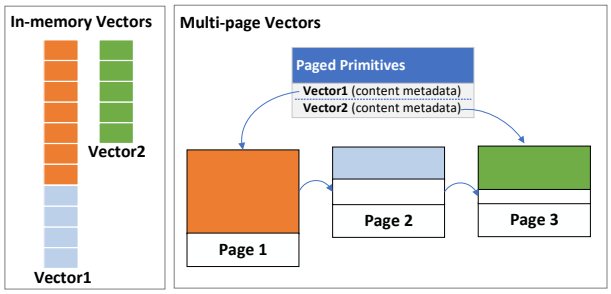

Figure 2: The multi-page vectors primitive: 2 in-memory vectors (left) and their byte-compatible paged primitives (right) on single page chain

5. PAGEABLE PRIMITIVES

Most of the common data structures of a column (i.e. data vector, dictionary, and inverted index) are implemented in HANA with the help of one or more compressed in-memory vectors. Vectors were chosen intentionally to store these data structures in contiguous memory. This enables optimized processor cache access to any element in constant time. This contiguous representation also enables optimized parallel scan operators that are fine-tuned to take advantage of multi-core vectorized processing [31]. The architecture of the native store extension is based on the unified persistency format and the ability for transient paged and in-memory structures to load from it efficiently. This requires not only to provide byte-compatible representation across in-memory and page loadable representation of every data structure, but also to employ identical algorithms to process classic and page loadable pieces of a column. This brings two key challenges: to provide paged counterparts of memory resident data structures, and to adapt existing algorithms to work seamlessly with either format without regressing HANA's column store performance. To achieve this, we introduced the concept of paged primitives. Primitives are key enabler of the unified persistency in NSE. We motivate the idea of composing primitives to support compressed pageable columns, as well as advanced memory reduction features, and remark on our performance enhancement.

5.1 Primitives

Paged primitives provide a uniform interface with their in-memory counterparts to read-write algorithms. This allows our code base to seamlessly operate on either format with minimum adaption, hiding the details of operating on data in native store extension. Furthermore, these primitives are composable and can be enriched with auxiliary and performant search structures and indexes.

5.1.1 *Multi-page Vectors*

A large vector can be stored on a single page chain with fixed size pages. Having a fixed number of objects per page simplifies identifying the page that has the content for a given vector position [29]. For NSE, we enriched this primitive with the possibility of storing more than one vector on each page chain (Figure 2), with each vector having its own metadata. Once a vector is sorted, each multi-page vector can be extended with a helper structure to facilitate search and to avoid loading pages that are guaranteed not to have a value that does not satisfy the search.

5.1.2 *Paged Mapped Vectors*

Unlike multi-page vectors, we may need to store many small vectors on a single page chain, and to have performant access to these small vectors. This is required by several compression schemes and features (Section 5.2 and 5.3). Such vectors can fit within a single page. During scans, a small vector is created directly pointing to memory of a loaded page, so it does not need to be copied, therefore reducing memory consumption.

5.1.3 *Arbitrary Sized Items*

Some data items are variable sized in nature, e.g. geometric objects (Section 5.3.1). Paging of such items requires a primitive to store blocks of variable size. A block is fully stored on a single page. The number of blocks stored on a page depends on block sizes. For data items larger than the page size, data is internally divided into multiple blocks. This primitive provides a key to its consumers when a new item is added. The key can be used to later retrieve the stored object.

5.2 Compressed Pageable Columns

HANA extensively uses advanced compression algorithms to reduce the memory footprint of columnar data and to achieve performance [18]. With the introduction of pageable columns [29], we mainly focused on supporting paging of data vectors compressed using uniform bit packing. While plain domain encoding provides good compression rate for data vectors, it could not exploit the increased savings achieved by the compression techniques that HANA uses for in-memory columns [18]. Consequently, the applications had to choose between an uncompressed pageable or compressed in-memory format. This restricts the data vector space that could benefit from the unified persistency (Section 4) to domain-coded columns. To tackle this limitation, we provide pageable counterparts to all compression formats supported by HANA. Beyond the obvious reduction in disk and memory footprint together with increased performance, this closes the gap between the page loadable and in-memory column and enables fast load unit conversion. In this section, we briefly overview three compression techniques and describe their page loadable counterparts.

5.2.1 *Pageable Run-Length Encoding*

The Run-Length Encoding (RLE) technique is suitable for a data vector with regions of repeated values (runs). It reduces this redundancy by representing each run by a pair of value identifier and frequency. In HANA we trade off some compression and prefer to store the start position of each run instead of the frequency. Therefore, a compressed data vector with RLE compression stores two vectors: the integer vector containing the identifier of values per run, and the start position of each run. Both vectors can become pageable by using paged mapped vectors. With RLE, we trade disk space with performance. The performance impact can be prohibitive when we want to access the value given a row position r; a brute force approach would load many compressed pages to locate the runs corresponding to r. To mitigate this problem, we introduced a compact memory resident helper structure that contains the last value recorded in each of the start positions vector pages and do binary search on this memory resident structure to narrow the number of pages loaded to two pages.

5.2.2 Pageable Sparse Encoding

When the value distribution of a column contains one very frequent value, in a simplified form, the compressed data can be represented by two vectors: an integer vector (the original vector with the most frequent value removed) and a bit vector with bits set at the positions where the most frequent value is observed. Pageable sparse encoding uses the multi-page vectors for storing the reduced data vector and a simplified multi-page vector for storing the bit vector that represents the position of the most frequent value.

5.2.3 Pageable Indirect Encoding

This encoding scheme can save more space by finding blocks in a data vector that have few distinct values. The compressed data vector is represented by an integer vector that keeps the block dictionaries and the uncompressed data. We store a vector of compressed blocks containing a pointer to the individual values and metadata. Pageable indirect encoding uses a multi-page vector to encode the integer vector that stores one dictionary per block. Each dictionary contains the distinct values observed per block. Each compressed block is stored using a small integer vector (a paged mapped vector) which can be loaded on-demand from the underlying pageable storage. We also store a vector of page logical pointers for each block (null pointer if a block is not compressed) followed by a vector of offsets in the value vector (one offset per block) using paged mapped vectors and arbitrary size items, respectively.

5.3 Advanced Memory Reduction Features

5.3.1 Paging of Geo-Spatial Columns

Geo-spatial columns typically store large data sets and can easily require large amounts of memory when loaded. Users are forced to choose between expensive hardware for hosting all of their geo-spatial data or to manage only a subset of the data. Paging of geo-spatial columns allow full HANA functionality over smaller memory footprints. Geo-spatial columns store one cell per block using the arbitrary sized items primitive (Section 5.1.3) and a block can span multiple pages when needed. The block address for each cell is stored separately using the multi-page vector primitive (Section 5.1.1). To scan a value for a row position, the multi-page vector primitive is used to retrieve the corresponding block address. Further, the block address is used to load the specified page using the arbitrary sized items primitive. Without paging, geo-spatial columns are loaded as a vector of reference counted objects. Each object points to corresponding geometry data, has size information, and a reference count which determines the life cycle of the data. Similarly, a paged geo-Spatial column cell is represented as a reference counted object where the object points directly to the geometry data in the corresponding page. The object also holds a handle to the underlying page to manage the page's life cycle.

5.3.2 Paging of RowID Column

HANA uses a persistent RowID column to uniquely identify rows across delta merges and similar operations which physically reorder rows for optimal storage. The RowID column is stored in compressed format where consecutive $1,024$ values are compressed together to form a block. We store such word boundary aligned blocks on the page chains. Alignment ensures that the blocks can directly be accessed from pages when needed without requiring them to be copied into some aligned temporary memory. The start position of each of the blocks is also stored at the end of the page chain, the so called block address vector. We use the multi-page vector primitive (Section 5.1.1) to store the block address vector. Considering the maximum number of rows per partition (2 billion), the block address vector is guaranteed to be not very large. During load of the column, the block address vector is fully loaded into memory from the pages. A frequent operation in HANA is to determine the position of a row containing a given RowID. This can be determined by using the block address vector to identify the page where the RowID is and its offset. Only the specified page is loaded and the blocks are accessed using the page mapped vectors' API (Section 5.1.2).

5.3.3 Paging Dictionary

In [29], we introduced on-demand dictionary access structures for variable size data types (e.g. VARCHAR). For fixed size data types (e.g. INTEGER) a sorted dictionary for the main fragment of a column is represented as a vector of fixed size values. There are two key operations on a column dictionary: 1) materialization of the value corresponding to the value identifier observed in a data vector at a desired row position, and 2) probing of the dictionary to perform reverse mapping (i.e. from value to its identifier in the data vector). For fixed size data types, we designed the paged dictionary using a variant of the multi-page vector primitive. To materialize a value, the primitive can provide a single page access guarantee. This is because each page can hold a fixed number of values. To probe a sorted dictionary, however, a binary search is required which may load many pages to locate the single page that contains the probed value. To mitigate this problem, we introduced a compact memory resident sorted vector (helper for dictionary). The last value on each dictionary page is stored in the helper. The probe operation can perform binary search on the helper and load a dictionary page only if there is a match.

5.3.4 Paging Hash-based Indexes

The hash-based indexes are unique multi-column indexes created over the table. Being unique, the dictionary is the major space consuming subcomponent of these indexes. We implemented a two-pronged approach to reduce the memory footprint. First, we replaced the dictionary in the hash indexes with a hash function and functionality for paging the row position mapping is provided through a hybrid hash column implementation. A variant of multi-page vector primitive (Section 5.1.1) is used to abstract paging. The hash values stored for the indexes are hidden and almost never projected through the user queries. Owing to this peculiarity of the hash-based index, the amount of index data accessed while executing a query is reduced. This removes the need to do page accesses for the dictionary lookup.

5.4 Remark on Performance Optimization

Paged primitives can be slower than in-memory counterparts, because of the requirement to access unloaded pages. Paged primitives are constructed for the main fragment of a column (the read-only, but large piece) as opposed to delta fragment. Large scans are typical on the main fragment.

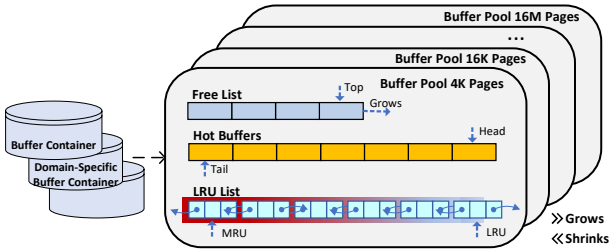

Figure 3: Architectural of HANA's buffer cache

To mitigate the natural impact of I/O latency when accessing paged primitives, a combination of optimization techniques can be used. First, as will be described in Section 7, HANA's execution engine can provide prefetch hints to the buffer cache for sequential scans that expose patterns when accessing primitives. Second, page pinning can prevent a trip to the buffer cache, by storing page handles in the context of the operator's stack to ensure repeated access to the page are not impacted by buffer cache eviction. Third, small memory resident data structures, e.g. page synopsis [4, 14, 24] can be designed to reduce the amount of page accesses and to cache statistics, as the main fraction of a column is guaranteed to not change between two consecutive delta merges.

6. HANA BUFFER CACHE

The introduction of the Native Store Extension in HANA necessitated a well performant and scalable buffer management system for page resources. HANA has a resource manager that manages different kinds of resources in memory and evicts them under high memory pressure situations using a weighted LRU strategy, where each kind of resource (columns, pages) can be associated with a specific weight. However, for better eviction strategies, we needed to further classify pages based on their access patterns. With this purpose, we have introduced a buffer cache which is maintained separately from the resource manager to manage the page resources within a memory limit. The buffer cache holds used and free pages and can grow and shrink on demand. Most used pages are retained in-memory while the least used ones are replaced, if needed, reducing HANA's overall memory footprint. HANA supports multiple page sizes based on their usage for data, dictionary, or index. Multiple buffer pools of different page sizes are maintained within the buffer cache. In this section, we explain features and requirements that motivated the design of the buffer cache architecture (Figure 3). HANA's buffer cache supports the creation of domain-specific page pools for objects (e.g. data, dictionaries, and indexes) helping to retain one set of heavily used pages more than the other, to improve system performance.

6.1 Buffer Cache Elasticity

The buffer cache supports all HANA page sizes by maintaining a buffer pool per size. On server start, the cache contains no memory. When a free buffer is requested, the buffer cache grows by pre-allocating free buffers in the background, so the subsequent requests for free buffers of the same page size need not wait on memory allocation. All the buffer pools can cumulatively grow up to the cache capacity. The growth size of each buffer pool can be tuned based on the frequency of the free page requests of its page size. Cache capacity can be tuned, and when reduced, the buffer pool will start asynchronously shrinking the cache by removing free buffers followed by unused pages proportionately across all the pools until the cache size falls below the reduced cache capacity.

6.2 Hot Buffer Retention and Page Stealing

Although the goal is to keep the memory footprint low by limiting usage through the limited buffer cache, the overall performance of the system should remain on par with that of the in-memory system. To achieve this, it is essential to reduce I/O by retaining frequently used buffers while maintaining enough free buffers for future usage. We provide an adaptive version of the LRU replacement policy, that not only provides the simplicity of the LRU mechanism, but also solves the synchronization overhead associated with it. We maintain a separate list of working set buffers in a lockless queue. A housekeeper thread monitors the overall buffer requests in the system and rebalances the buffers from the working set queue into the LRU list and free list. This also follows a policy of retaining frequently used buffers, extending the stay of hot buffers in the cache until there is an occasional burst of buffer requests e.g. by delta merge operation. As buffers exist in different pools, a pool size balance is maintained by stealing buffers from pools with least activity. This is critical for the frequent scenario of a specific pool being heavily used causing LRU replacements leading to a reduced performance due to page I/O.

6.3 Page Prefetch

The buffer cache allows queries to asynchronously prefetch pages before accessing them to reduce wait on I/O. Every prefetched page follows the same life cycle of an LRU page. For example, analytical queries access multiple pages of each column, and many such queries could run concurrently. This can result in many concurrent page prefetch requests. However, buffer cache prefetch subsystem limits the total number of prefetched pages to certain percentage of the cache capacity so the synchronous page loads or page allocations do not starve for free buffers. To fulfill large prefetch requests, despite a limited prefetch size, the prefetch subsystem allows the queries to incrementally prefetch a list of pages. Queries run with multiple jobs, and before jobs for a query begin, the first few pages of the range of pages each job will access are prefetched. As each job begins its processing, the remaining pages are prefetched.

6.4 Out-of-buffers and Emergency Pool

Queries fail if the buffer cache fails to find a free buffer despite a reasonable number of retries. In this case, the user is alerted to increase the buffer caches capacity. But some critical tasks like undo of a transaction, crash recovery, or continuous log replay on a secondary site, cannot fail as their failure would leave the database in an inconsistent state. For these cases, a free buffer request is fulfilled by an overflow resource provider managed outside the buffer cache. Here, the free buffer provided for the page is destroyed immediately after the page has been used for its designated purpose. By allocating-on-demand and immediate destruction of the emergency buffer, the memory consumed by the overflow resource provider is kept as low as possible.

7. HEX INTEGRATION

An incoming query in HANA is first parsed and optimized by the SQL frontend. In this step, various static rules (e.g. convert expressions to conjunctive normal form, eliminate constants and tautologies, apply static partition pruning) and cost-based heuristics (e.g. enumerate different join orders, pick plan alternative with lowest estimated cost) are applied to produce an optimized logical execution plan. The logical operators are subsequently substituted by physical operators (e.g. replace a generic join by a hash, nested, or loop index join). If a physical HEX alternative exists for each operator, HEX is used for execution, else the query is sent to one of HANA's classical query engines.

7.1 Query Execution in HEX

Relational databases traditionally evaluate a physical operator tree in a root-to-leaf fashion using the iterator model first popularized by the Volcano system [13]. In that model, parent operators repeatedly pull single tuples from their child operators by calling a virtual *getNext()* method. Volcano style processing is simple and versatile and had its merits in a time when disk I/O was the bottleneck. However, *getNext()* is not only invoked very frequently (once per tuple and operator), also each call needs to be resolved in a *vtable*. This makes the iterator model generally ill-suited to the branch prediction of modern CPUs. HEX instead adopted a data-centric, leaf-to-root push model [23, 21]. In this approach, the control flow is reversed: operators push tuples towards their parent operators using *produce()* and *consume()* methods. Neighboring HEX operators are said to form a pipeline if they pass batches of tuples, as many as fit the Lx cache, without spilling them to DRAM or to disk. To achieve maximum flexibility, HEX operators can add two kinds of code to a pipeline. One kind inserts precompiled code which usually wraps complex logic like decompression of domain-coded tuples or disk pages fetches. The other kind emits code in a high-level programming language (L, an SAP-internal language) which is by default either interpreted upon the first query invocation or, upon repeated invocation, compiled to native code at runtime using LLVM. Adjacent generated operators can be amalgamated into a single, fused operator. The resulting tight nested loop greatly benefits code locality and data locality as tuples typically reside fully in CPU registers during execution.

7.2 Column Scans with Prefetch

Some of the most I/O-intensive operations in many HEX plans are pipelines which are fed by a main fragment scan. Such pipelines are typically generated from WHERE clauses in SQL queries with predicates on one or more columns. A main fragment scan comprises two steps. In the first step, called dictionary lookup, the scan predicate is translated into a set of value ids using the dictionary. For most predicates, this breaks down to (few) binary dictionary searches with logarithmic runtime. The second step, called data vector scan, compares each value id of the data vector against the value ids found in the previous step. Data vector scans tend to be more expensive than dictionary scans because the size of the data vector usually exceeds the dictionary size by far and, the scan advances row-by-row with linear runtime. To that end, the HEX framework breaks up the scan range into subranges, and eventually scans the subranges using multiple threads in parallel. If the scanned column is paged and not loaded, scan jobs will repeatedly trigger blocking page loads via the buffer cache. To avoid such unnecessary roundtrips to persistency, a prefetch pipeline is created and populated with prefetch operators, one per column fragment, and run as early as possible. The prefetch operator requests the buffer cache to preload as many pages of the respective subrange as possible. Prefetching generally has best-effort semantics, i.e. if the page cache area is already highly utilized or if the I/O system is contended, the buffer cache may decide to drop parts of the prefetch request. We found that fairness is a desirable property of prefetching. By that we mean that approximately the same number of pages shall be loaded for all subranges and that pages shall be loaded in the same order in which the scan visits them. If prefetching is not fair, some scan jobs are given a head start over other scan threads and finish (in average) earlier than penalized threads. This works in general against the scalability and resource utilization of the database system. To ensure fairness, our implementation of prefetch asks one subrange after the other to advance a per-subrange row watermark by which the scan can proceed, assuming it can access a constant number of pages. Prefetch stops if the buffer cache prefetch limit has been exhausted or if all subranges have been prefetched. During the data vector scan, a best-effort attempt will be made to prefetch any remaining pages in each subrange not accepted during the first prefetch request.

Algorithm 1 GetLoadUnitAffinity

```
1: if column load unit hint ≠ "default loadable" then
2:     return column load unit hint
3: else if partition load unit hint ≠ "default loadable" then
4:     return partition load unit hint
5: else if table load unit hint ≠ "default loadable" then
6:     return table load unit hint
7: else
8:     return "column loadable"
```

8. LOAD UNIT CONFIGURATION

For HANA to use the features introduced so far, database objects (tables, partitions and columns) need to be configured to use NSE. This section describes the metadata and SQL interface used to configure NSE, as well as the process of load unit conversion, which is required to convert existing database objects to use the unified persistency format.

8.1 Persistent Metadata and SQL Interface

HANA provides the functionality to either page individual columns or to use the data aging feature. In data aging, a single partition will be in-memory and all other partitions paged [29]. HANA's NSE adds flexibility by providing a SQL interface to tag columns, partitions and tables with load unit hints and convert the underlying column persistency to the preferred storage format. The load unit hint tag is persistent and acts as a hint to determine the loading behavior of columns i.e. in-memory or paged. Column, partition and table can be tagged with one of the three load unit hints: column-loadable to indicate the columns to be loaded as fully in-memory, page-loadable to indicate columns to be loaded as paged, default-loadable to indicate the absence of any explicit load unit hint. The desired loading behavior for a column in a given partition is derived during the load unit tagging operation as described in Algorithm 1.

8.2 Online Load Unit Conversion

Prior to NSE, HANA used a different persistency format for in-memory vs. page loadable column structures [29]. This is because in-memory structures do not require page alignment, so the most efficient way to store them was simple serialization. NSE hybrid column structures use a page-aligned unified persistency format (Section 4). This means tables with persistency created prior to NSE need to convert their persistency to the unified format to take advantage of the hybrid column store features. This persistency conversion is termed load unit conversion and is done through an explicit SQL alter column DDL.

In HANA, table columns have a numeric identifier which is unique within a table and consistent across all partitions of that table. Before the introduction of NSE, alter column operations in HANA were implemented by internally creating a new table column with a new identifier followed by copying data from the old column to the new column and subsequently dropping the old column. DDLs to alter the load unit of a given partition, which internally uses the alter column infrastructure, would have to operate on all the partitions of the table to guarantee a consistent column identifier across partitions. HANA also uses internal columns with fixed unique column identifiers where load unit conversion must be supported. With NSE, the online load unit conversion within a partitioned table is no longer done by introducing a new column, but by versioning existent columns while keeping their identifiers. The load unit conversion DDL creates a new version of the column with the preferred persistency format, while existing readers referring to the old column version can continue to access it. The new column version is activated when the DDL commits, from which time readers use the new column version. This column versioning mechanism allows HANA to execute load unit conversions on specific partitions while preserving the same unique column identifier across partitions.

8.3 Partitioning with NSE

Unbalanced partitioning is a flexible partitioning strategy that departs from the conventional balanced partitioning scheme. It allows to define separate partitioning schemes (e.g. hash) at selected nodes of the second level of partitioning subtrees than the first level which uses range partitioning. For instance, one could decide on a range-range partitioning scheme, and have different ranges at the 2nd level for some of the 1st level ranges, or not even have a range at the 2nd level for some of them. This scheme is very useful in organizing the table based on data temperature i.e. based on the hot vs. cold nature of the data. It may be possible, for instance to organize the cold partitions differently as the access requirements are optimized by design. The partitioning specification for such a scheme is internally represented as a JSON document; very different from a linear text string used in balanced partitioning.

The partitioning schema offers the flexibility to declare specific properties assigned for each physical partition, e.g. the location, the load unit specification, and the partitioning group types. Declaring inheritance rules allows to inherit from parent level (i.e. 2nd level can inherit from 1st level). Furthermore, one can partition at the second level using a different partitioning statement, e.g. *range(Column1)* and *range(Column2)* for some of the 1st level ranges but *range(Column1)* and *range(Column3)* for some other partition nodes. This enables specializing and balancing the partitioning tree based on actual data distribution. Unbalanced partitioning integrates well with the Native Store Extension. As opposed to classic data aging, where each partitioned table could have one hot current partition and a set of cold aged partitions, we can designate selected nodes in the partitioning tree to be page loadable with NSE. This ensures that the data for all the columns in those partitions would reside on disk and will be paged via the buffer cache when the data is queried. This relaxes the memory requirements for storing very large tables in HANA. The NSE advisor can analyze workloads on partitioned tables and recommend the change of the load unit for each partition node.

9. LOAD UNIT ADVISOR FOR NSE

To achieve the best cost/performance ratio, it is necessary to know which objects should be made page loadable to reduce total memory footprint without significantly affecting the performance. This section describes a solution to help identify the objects (tables, partitions, or columns) that are suitable to be converted to page loadable (to save the memory space) or to column loadable (to improve performance). Ideally, objects that are small and accessed frequently should be kept in memory to improve column data access performance. Large objects rarely accessed are better to be kept on disk to reduce memory usage, and only brought in memory when needed and only for the pages that are accessed. Our solution consists of two phases: access pattern collection and rule-based load unit recommendation.

9.1 Access Pattern Collection

To build the data access pattern for a workload, each physical access to a column fragment is counted and updated at query level in an internal cache unit called the access statistics cache. Becuase only the main fragment is paged for a page loadable column, the access count is only recorded for accesses to the main fragment. When the delta fragment is merged to the main fragment, the access count for the original main fragment is retained. When a column is unloaded from memory during the workload, its access count is also retained in the access statistics cache and continues to be updated after the column is loaded again into memory. DDLs that drop an object will clean up the statistics entries related to the dropped objects. Building the data access statistics has a small impact on the ongoing workload performance. Therefore, it is only enabled through a configuration option. The statistics cache can be cleared when not needed or when a new cache needs to be built.

9.2 Object Load Unit Recommendation

Load unit recommendations uses a heuristics rule-based approach that relies on measurements and thresholds:

Scan density $d(o)$ ratio of access scan count of object o over the object's memory size

Hot object threshold θ_h minimum scan density for an object to be considered a hot object

Cold object threshold θ_c maximum scan density for an object to be considered a cold object

Object size threshold θ_s minimum object size to be considered for recommendation

The thresholds (θ_h, θ_c, and θ_s) are system parameters. A list of objects is recommended to convert their load units

based on their scan density. For object o[1], if $d(o)<\theta_c$ and the object's memory size is greater than θ_s, it is recommended to be page loadable. If $d(o)>\theta_h$, it is recommended to be column loadable. A table may be recommended to be page loadable due to an overall low scan density while a column within the table may be recommended to be column loadable due to its high scan density. In that case, the column level recommendation takes precedence over the table level recommendation. Similarly, partition level recommendation takes precedence over the table level recommendation.

10. EVALUATION

We report the performance analysis and the memory footprint reduction for SAP HANA. We compare default column loadable tables, where columns are completely loaded into memory (CL) vs. columns that use the Native Store Extension (NSE). We report only the results from selected set of experiments for brevity.

10.1 Workload and Metrics

We used an in-house benchmark tool (ML4). ML4 simulates production environment SQL workload at customer site with configurable scale factor. ML4 simulates users' interactions on an ABAP stack based on SAP S/4HANA application. ABAP is server programming platform for SAP applications. It sends SQL statements to the database. Basis of the workload is a sell from stock process including creation of sales orders, outbound deliveries, goods movements and billing documents (OLTP part) with reporting queries (OLAP part). A single scenario loop in ML4 simulates business transactions consisting of multiple user interaction steps. Each step is separated by an artificial constant waiting time of 10 seconds (think time). A typical test setup consists of multiple concurrently simulated scenarios executed by different users, where each user runs a series of loops. The first few set of transactions sometimes includes the execution of a reporting query, hence creating the OLAP part of the workload. The remaining transactions form the OLTP part. When simulating a certain number of users, OLTP and OLAP loads are balanced by default, allowing statistically 1 out of 100 users to perform a reporting query in the first transaction. The underlying HANA database used by ML4 has at least 100,000 tables with every column type that is used in a typical SAP S/4HANA installation, with different compression format (i.e. Clustered, Indirect, Prefixed, RLE, Sparse) as well as no compression. We conducted two sets of experiments: CL when accessed columns were completely loaded, and NSE when the majority of tables were converted to use Native Store Extension. In NSE experiments, we turned on the Load Unit Advisor for NSE (Section 9) to collect statistics from the workload and to provide recommendation on converting columns to NSE format. We report two set of metrics; memory consumption and performance. Memory consumption was measured at two levels; total system memory (M1) and the aggregate of memory footprint for tables (M2). The latter metric excludes the memory not used by the column store, e.g. for storing intermediate results of queries. We report performance as average total database time for running OLTP and OLAP transactions (we call these OLTP_MMV and OLAP_MMV, where MMV stands for ML4-metric value). We restrict the amount of memory used by

[1]a table, a partition, or a column

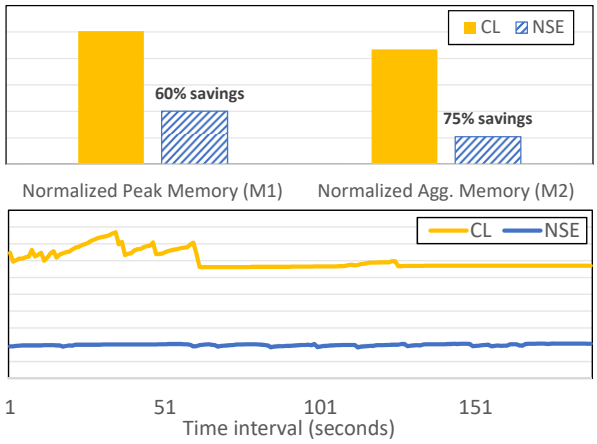

Figure 4: Normalized memory (OLTP workload)

HANA server to around 2X the size of the database, and we conducted NSE experiments by fixing the buffer cache size at 10% of data size as default. We also report experiment varying the buffer cache size.

10.2 Results and Analysis

Figure 4(top) presents the normalized memory measurements (M1 and M2) for running OLTP workload. Both CL and NSE benefit from compressing index vectors, but NSE does not need to load each column accessed for each OLTP operation completely into memory. Because of this, we observed immediate memory savings for both measures. We also monitored the memory consumption of the two settings (M2) over time, shown in Figure 4(bottom). For NSE, the memory consumption is restricted by the buffer cache size, whereas for CL, it is limited by the total memory available to HANA server. Notice that there are fluctuations for CL somewhere around 40 seconds. These correspond to accesses to column(s) designated as CL that had not been loaded into memory yet when the benchmark executed, and the read only portion of these columns need to be accessed, e.g. for checking a uniqueness constraint, or for system level statistics collection. As the buffer cache is operating at its full capacity, the same operation would trigger page evictions, to accommodate new requests. This kept the total memory consumption for NSE balanced in this experiment. This is a very desirable property for cloud deployment. The majority of the OLTP requests that require modifications of any type (INSERT/UPDATE/DELETE) were handled by the delta store of each column. Because the NSE technology only targets the read optimized section of column store (i.e. the main store), the performance of OLTP operations were not impacted significantly (less than 3.7%) by choosing the NSE configuration over fully loaded setting.

We conducted experiments with a mixture of OLTP and OLAP operations and observed similar trends as in the OLTP experiments for normalized runtime (Figure 5). The buffer cache again guaranteed that the total memory usage remains smaller than the amount of memory consumed by the CL setting (Figure 6 top) with a system memory usage pattern (Figure 6 bottom) very similar to the OLTP only experiments in Figure 4 bottom. Converting to NSE does not significantly affect the runtime, as shown in Figure 7. We should note that in the extreme case, when the buffer

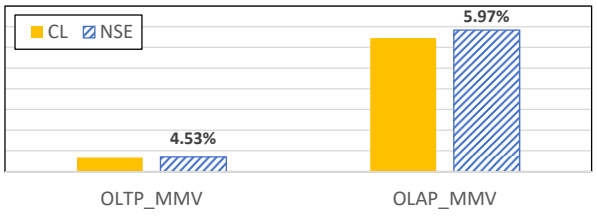

Figure 5: Normalized runtime (mixed workload)

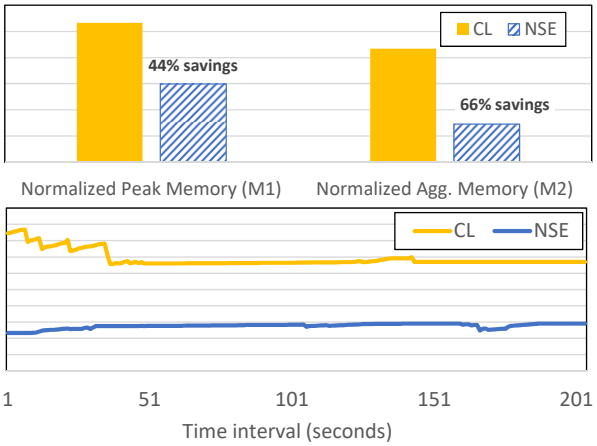

Figure 6: Normalized memory (mixed workload)

cache is very small, any operation that requires a portion of data which is not in the buffer cache would incur I/O to load required page(s). However, there is a significant difference with when a needed column is not present in the memory; NSE only loads sections of a column (e.g. pieces of compressed index vector, fractions of dictionary, and inverted index). However, CL needs every subcomponent of a column to be entirely present in memory. Because of this, CL may suffer more from column evictions, whereas NSE's granularity of eviction from the buffer cache is page units.

To evaluate the impact of buffer caches size, we conducted both OLTP and OLAP experiments, varying the size of buffer cache (Figure 7 and 8). NSE benefits from the NSE advisor integrated with the buffer cache, to enhance the page eviction decisions, and to initiate prefetch requests if needed. Because HANA's memory resources for CL setting is managed separately by HANA's resource manager, as opposed to NSE buffer cache, we did not observe difference in memory consumption and runtime for CL. When the size of the buffer cache reduces, we see the impact on the runtime for both OLTP and OLAP, with OLAP being more affected because it accesses more pages (Figure 7). This is expected, as buffer cache is pressured to evict more pages when the size decreases. This translates into more page misses in the buffer cache, and less average page lifetime. Reducing buffer cache size has no impact on peak and average memory for CL, but directly impacts NSE (Figure 8).

We conducted another experiment by reducing the global allocation limit (GAL). This limit restricts the amount of memory available to CL objects, and it has no impact on NSE resident objects. Reducing GAL imposes memory pressure on the HANA server, which results in the resource manager

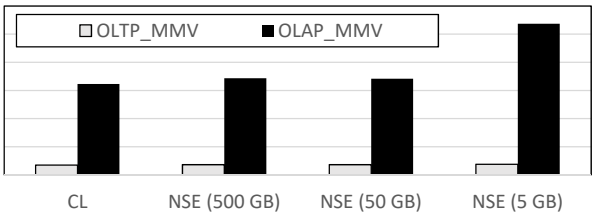

Figure 7: Normalized runtime varying buffer cache size (mixed workload)

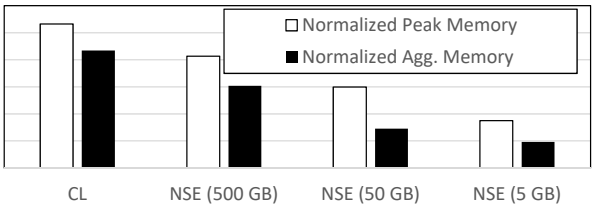

Figure 8: Normalized peak memory varying buffer cache size (mixed workload)

Figure 9: Normalized peak memory (top) and normalized runtime (bottom) varying global allocation limit (OLAP workload)

evicting database objects, at table and column granularity, to free memory. This eviction is orthogonal to the page eviction by the buffer cache. The former type of eviction results into reduction in normalized peak memory (Figure 9 top) and increased memory traffic, which impacts normalized query response time in CL (Figure 9 top). The buffer cache size is not affected by the total memory reduction, as long as it does not affect the amount of resources it preallocates. Hence, no performance impact for NSE (Figure 9 bottom).

11. RELATED WORK

We focus on selected components of NSE and place our contributions in the context of the state of the art.

Columnar databases use compression techniques extensively [2] and HANA implemented highly optimized lossless packing techniques for its in-memory column store [18]. Pairing paging and compression is a challenging problem and has been studied for data pages [27, 8]. Storing geo-spatial

data in paged format has been implemented before, with integration with specialized query processing engines [10, 32]. In our work, we proposed primitives that were designed for generic use case, and can be used to provide pageable compression and pageable geospatial columns. The byte compatibility with in-memory columns provided seamless integration with query processing at upper layer, without the need to specialize highly optimized algorithms.

Enterprise-class main memory translytics systems have started focusing on cloud-native infrastructure as a service to balance TCO. For instance, Oracle [1] uses a disk-based row store backed by a buffer cache as the warm store. The in-memory store with compression units support fast access to the columnar hot store. The row store is backed by a buffer cache to balance TCO and performance. These separate two layouts require format conversions and data migration to move data to a slower volume. Our unified persistency format mitigates this problem and enables dynamic load unit conversion to keep TCO at a desired level for cloud deployment. Stoica et. al. [30] use OS paging directly to efficiently migrate cold data efficiently to secondary storage by relying on the virtual memory paging mechanism of the operating system. Unlike NSE, their focus is only on row stores and OLTP. As this approach does not need buffer cache, it is hard to benefit from the optimizations possible by maintaining statistics from query execution engine at lower granularity, e.g. primitives. SQL Server [15] maintains a columnar store index (CSI) on memory-optimized row store tables for operational analytics on hot data. Operational analytics on warm data is done by creating and maintaining secondary CSI on disk-based row store tables. This results in additional overhead of maintaining the dual stores and moving data.

The use of buffer pools as a performance optimization component dates back to the early days of database systems [28]. Several modern in-memory databases use a form of buffer pool [5]. New hardware technology has been recently adapted to enhance the performance of classic buffer pool. Sai et. al. [25] propose a new replacement policy that separates the queue of managed resources into clean and dirty pools, when the database is stored on flash disk. Do et. al. [9] propose alternative eviction policies enhanced by high I/O bandwidth of SSD drives and predicted block access patterns. Liu and Salem [19] propose cost aware replacement algorithms for hybrid storage configuration (SSD and HDD drives). Arulraj et. al. [7] propose multi-tier buffer management using DRAM, NVM, and SSD that can be used for different workloads using different storage hierarchy for optimum performance. Graefe et. al. [12] propose a design that uses pointer swizzling to eliminate buffer pool overheads for memory resident data, in which raw pointers to loaded pieces are kept as reference to avoid lookup in separate queues managed by the buffer pool. LeanStore [17] uses pointer swizzling and replacement strategy by identifying infrequently accessed pages and an epoch-based technique to pin pages and achieve performance comparable to main memory systems. In HANA's NSE, the read optimized section of a hybrid column benefits from the NSE architecture and is managed by NSE's buffer cache for better tuning and optimizations that are only applicable to paged resources. Our buffer cache was designed from the ground up to take advantage of lessons learned from SAP's ASE [13] and HANA and providing extensibility as well as integration with HANA's brand new execution engine HEX, and NSE's hybrid column and primitives. NSE uses advanced replacement strategies to retain frequently accessed pages while maintaining balance of buffers for reuse across different page size pools. HANA's page accesses avoid the hash lookup by caching the buffer pointers in the page chain metadata.

The design and configuration of individual database features is a challenging task that significantly affects the performance and the total cost of ownership. The database community has studied this problem in the context of self-management for relational databases [11], to recommend the design of physical layout [19, 33], to create new set of indexes [3, 33], to define new data statistics objects [24], to alter partitioning landscape [22], or to tweak the knobs and switches [3], all with the common goal of running the database at its highest performance level. The combination of NSE's hybrid column store and partitioning proposes the opportunity to balance TCO with optimum performance goals. With this goal in mind our advisor pairs with byte-compatible primitives to support dynamic load unit conversion unique to NSE and to maintain a desired balance between performance and TCO.

12. SUMMARY

We presented the architecture of HANA's Native Store Extension, with a unified persistent format and pageable primitives at its foundation. By integrating NSE into HANA's in-memory columnar store we support hybrid columns with advanced paged compression and reduced memory requirements. Hybrid columns, being a primary citizen of HANA's column store, require a new buffer cache to effectively manage pageable resources. This buffer cache works in conjunction with HANA's resource manager which oversees the in-memory structures. The buffer cache was optimized considering the nature of hybrid columns and page access patterns, and for tight integration with HANA's HEX query execution engine. HEX can feed the buffer cache with hints to enhance its operation, e.g. buffer retention policies, among many other optimized algorithms that we implemented already. Through comprehensive end-to-end experiments, we have shown that on typical customer scenarios NSE reduces the memory footprint with a minimal loss of performance compared to in-memory column stores, especially when the required resources for a query are provided by the buffer cache. Our approach proved to be fruitful. Beyond its initial goal, NSE and the unified persistence format in conjunction with a flexible load unit configuration and partitioning enables several desirable applications, including data tiering, K-safe high availability with reduced TCO, and a conventional disk database with fast restart.

13. ACKNOWLEDGMENTS

We thank the anonymous reviewers for the comments and suggestions which contributed to improving the presentation of this paper. We give special thanks to colleagues and the members of NSE, Store, HEX, Attribute Engine, Basis and Persistency, and SPeeD teams for their contributions to the NSE project, and to Tony Zhou for contributing to Section 9.

14. REFERENCES

[1] Oracle Database 19c In-Memory Guide. https://docs.oracle.com/en/database/oracle/oracle-database/19/inmem/, 2019. [Online; accessed 03-February-2019].

[2] D. Abadi, P. Boncz, and S. Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., 2013.

[3] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *ACM SIGMOD*, pages 1009–1024, 2017.

[4] K. Alexiou, D. Kossmann, and P.-Å. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB*, 6(14):1714–1725, 2013.

[5] T. Anderson. Microsoft SQL Server 14 man: Nothing stops a Hekaton transaction. http://www.theregister.co.uk/2013/06/03/microsoft_sql_server_14_teched/, 2013. [Online; accessed 03-February-2019].

[6] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA Adoption of Non-Volatile Memory. *PVLDB*, 10(12):1754–1765, 2017.

[7] J. Arulraj, A. Pavlo, and K. T. Malladi. Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. *CoRR*, abs/1901.10938, 2019.

[8] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient Index Compression in DB2 LUW. *PVLDB*, 2(2):1462–1473, 2009.

[9] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. Naughton, and A. Halverson. Turbocharging DBMS Buffer Pool Using SSDs. In *ACM SIGMOD*, pages 1113–1124, 2011.

[10] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial Partitioning Techniques in SpatialHadoop. *PVLDB*, 8(12):1602–1605, 2015.

[11] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical Database Design for Relational Databases. *ACM TODS*, 13(1):91–128, 1988.

[12] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. In-memory Performance for Big Data. *PVLDB*, 8(1):37–48, 2014.

[13] A. Gurajada, D. Gala, F. Zhou, A. Pathak, and Z. F. Ma. BTrim: Hybrid In-memory Database Architecture for Extreme Transaction Processing in VLDBs. *PVLDB*, 11(12):1889–1901, 2018.

[14] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *ACM SIGMOD*, pages 311–326, 2016.

[15] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-time Analytical Processing with SQL Server. *PVLDB*, 8(12):1740–1751, 2015.

[16] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *ACM SIGMOD*, pages 1307–1318, 2016.

[17] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-Memory Data Management beyond Main Memory. In *IEEE ICDE*, pages 185–196, 2018.

[18] C. Lemke, K. Sattler, F. Faerber, and A. Zeier. Speeding Up Queries in Column Stores - A Case for Compression. In *DAWAK*, pages 117–129, 2010.

[19] X. Liu and K. Salem. Hybrid Storage Management for Database Systems. *PVLDB*, 6(8):541–552, 2013.

[20] N. May, A. Böhm, and W. Lehner. SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *BTW*, pages 545–563, 2017.

[21] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *PVLDB*, 11(1):1–13, 2017.

[22] R. Nehme and N. Bruno. Automated Partitioning Design in Parallel Database Systems. In *ACM SIGMOD*, pages 1137–1148, 2011.

[23] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[24] A. Nica, R. Sherkat, M. Andrei, X. Cheng, M. Heidel, C. Bensberg, and H. Gerwens. Statisticum: Data Statistics Management in SAP HANA. *PVLDB*, 10(12):1658–1669, 2017.

[25] S. T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu. FD-buffer: A Buffer Manager for Databases on Flash Disks. In *ACM CIKM*, pages 1297–1300, 2010.

[26] H. Plattner. The Impact of Columnar In-memory Databases on Enterprise Systems: Implications of Eliminating Transaction-maintained Aggregates. *PVLDB*, 7(13):1722–1729, 2014.

[27] M. Poess and D. Potapov. Data Compression in Oracle. In *VLDB*, pages 937–947, 2003.

[28] G. M. Sacco and M. Schkolnick. A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model. In *VLDB*, pages 257–262, 1982.

[29] R. Sherkat, C. Florendo, M. Andrei, A. Goel, A. Nica, P. Bumbulis, I. Schreter, G. Radestock, C. Bensberg, D. Booss, and H. Gerwens. Page As You Go: Piecewise Columnar Access In SAP HANA. In *ACM SIGMOD*, pages 1295–1306, 2016.

[30] R. Stoica and A. Ailamaki. Enabling Efficient OS Paging for Main-memory OLTP Databases. In *ACM DaMoN*, pages 7:1–7:7, 2013.

[31] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing Database Column Scans with Complex Predicates. In *ADMS@VLDB*, pages 1–12, 2013.

[32] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient In-Memory Spatial Analytics. In *ACM SIGMOD*, pages 1071–1085, 2016.

[33] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004.