An Approach for Hybrid-Memory Scaling Columnar In-Memory Databases

Bernhard Höppner
SAP SE
Potsdam, Germany
bernhard.hoeppner@sap.com

Ahmadshah Waizy
Fujitsu Technology Solutions GmbH
Paderborn, Germany
ahmadshah.waizy@ts.fujitsu.com

Hannes Rauhe
SAP SE
Potsdam, Germany
hannes.rauhe@sap.com

ABSTRACT

In-memory DBMS enable high query performance by keeping data in main memory instead of only using it as a buffer. A crucial enabler for this approach has been the major drop of DRAM prices in the market. However, storing large data sets in main memory DBMS is much more expensive than in disk-based systems because of three reasons. First, the price for DRAM per gigabyte is higher than the price for disk. Second, DRAM is a major cause for high energy consumption. Third, the maximum amount of DRAM within one server is limited to a few terabytes, which makes it necessary to distribute larger databases over multiple server nodes. This so called *Scale-Out* approach is a root cause for increasing total costs of ownership and introduces additional overhead to an in-memory database system landscape.

Recent developments in the area of hardware technology have brought up memory solutions, known as Storage Class Memory, that are capable of offering much higher data density and reduced energy consumption than traditional DRAM. Of course these advantages come at a price: higher read and write latencies than DRAM. Hence, such solutions might not replace DRAM but can be thought of as an additional memory-tier. As those solutions are not available on the market today, we investigate the usage of an available bridge-technology combining high-end flash SSD and DRAM referred to as *hybrid-memory*.

In this paper we present an approach for columnar in-memory DBMS to leverage hybrid-memory by vertically partitioning data depending on its access frequency. We outline in what way our approach is suited to scale columnar in-memory DBMS to overcome existing drawbacks of Scale-Out solutions for certain workloads.

1. INTRODUCTION

In previous years computer architecture distinguished between two types of data mediums, storage and memory. This separation likewise defined the design of Database Management Systems (DBMSs), making use of Dynamic Random Access Memory (DRAM) as a low latency and therefore fast but at the same time expensive as well as density-limited cache. Storage – mainly Hard-Disk-Drives (HDDs) – became the central store of the database [7].

Recent development in hardware has lead to rapidly dropping market prices of DRAM in the past years. This development made it economically feasible to use DRAM as the primary data store of DBMS which is the main characteristics of an in-memory DBMS [21]. In this paper we focus on *SAP HANA*, a columnar in-memory DBMS released early in 2010 by SAP [26]. Further commercial systems are *VoltDB* [28], *IBM BLU* [23], *Microsoft Hekaton* [17] as part of SQLServer 2014 and *Oracle TimesTen* [16]. Research systems such as *Hyper* [15] and *Hyrise* [12] are also focusing on in-memory storage.

Major drawbacks of using DRAM as the main data medium are the physical limitations in terms of size as well as the cost gap between memory and storage. Not only is the price per gigabyte (GB) much higher for DRAM than for Solid-State-Drives (SSDs) and HDDs, it is also a major driver of energy consumption because of its frequent refresh cycles [32]. Also the amount of DRAM in a single server is limited by the maximum amount of Central Processing Units (CPUs) nodes in a machine, as in today's systems memory controllers are directly attached to the CPU dies. Those architectural drawbacks of DRAM cannot be overcome, although the maximum amount of DRAM in a single server is increasing, for example because of an increasing number of available Dual Inline Memory Module (DIMM) slots per CPU or by introducing proprietary Non-Uniform Memory Access (NUMA) switches [25, 9]. SAP HANA can be distributed over multiple nodes using a shared nothing approach to manage the limited capacity of DRAM per server [18]. Today this so called *Scale-Out* is the only option available if it is necessary to store databases exceeding the memory limit of a single server. This leads to an overhead as the distribution of transactions between nodes needs to be handled and synchronized by the DBMS. Additionally, the Total Cost of Ownership (TCO) of the system landscape increases significantly due to higher efforts for administration and operation of the necessary hardware and the more complex infrastructure. Furthermore, many scenarios may not require additional computation power. In fact, the memory capacity of a single server is exceeded because data cannot be dropped, e.g., for legal reasons, from the database even

Proceedings of the VLDB Endowment, Vol. 7, No. 14

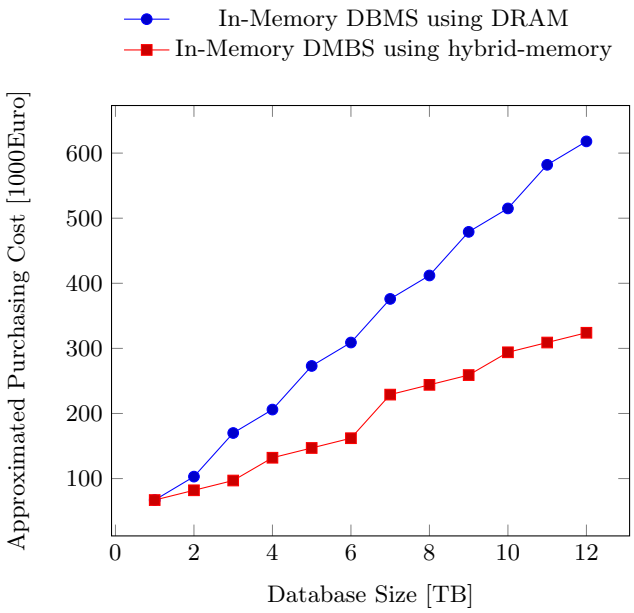


Figure 1: Approximated purchasing cost of in-memory DBMS and in-memory DBMS using hybrid-memory. DRAM to hybrid-memory ratio not smaller than one to two.

though it is accessed rarely.

We believe that upcoming memory technologies are well suited to offer an additional approach for scaling columnar in-memory DBMS in a way which is capable of overcoming the mentioned issues using a simple method for vertical data partitioning. Storage Class Memory (SCM) is a term used to address such memory technologies, referring to high dense, low latency, and persistent memory which closes the gap between memory and storage. The terminology has first been used by IBM [8] and refers to a variety of different hardware solutions such as Phase Change Memory, Spin Transfer Torque RAM, Magnetic RAM, and Memristors which are all sharing the characteristics mentioned to a certain degree [2]. These technologies are no commodity hardware today but will become reality in the near future. In the meantime the expected latencies can be emulated as proposed in [14] to gain first insights into the effects of using SCM. In this paper we used a bridge-technology designed by Fujitsu Technology Solutions GmbH combining high-end SSDs and DRAMs into hybrid-memory to be able to test some of the characteristics of future SCM and concentrate on the high memory density compared to DRAM.

The bridge-technology relies on latest SSDs technology as data medium which is buffered by traditional DRAM. The concept used is similar to the paging algorithms implemented in the Linux operating system [1]. Additionally, our solution offers several new features and configurations such as limiting the available DRAM buffer, introducing a predefined read-ahead functionality, as well as early page eviction. Especially the limitation of the buffer size makes it feasible to use hybrid-memory as an *additional* memory-tier next to DRAM. From a business perspective hybrid-memory is well suited to reduce the costs of scaling a database system as approximated in Figure 1 because of reducing the number of servers and amount DRAM needed. Hence, it is also capable of reducing energy costs, downtime and maintenance expenses. Depending on data access patterns and the size of the hybrid-memory buffer a slowdown of data access can be assumed when introducing the technology into an in-memory DBMS. Therefore, the data needs to be partitioned depending on its access characteristics so that frequently accessed data is kept in DRAM whereas seldom used data can be moved to hybrid-memory.

The focus of this paper is to:

- Extend a columnar in-memory DBMS by offering an additional data store which can be used for rarely accessed columns generating no overhead for in-memory columns.
- Present an approach for vertically partitioning a columnar DBMS utilizing a slower but denser memory technology.
- Evaluate the vertical data partitioning approach based on the trace of a productive DBMS and a commonly known benchmark.

2. VERTICAL PARTITIONING

In this paragraph we introduce our approach for columnar in-memory DBMS to leverage hybrid-memory as explained in Section 3. First we present our approach for vertically partitioning data for hybrid-memory. Afterwards, we analyze data access patterns in a productive Enterprise Resource Planing (ERP) system to evaluate our idea.

2.1 Data Partitioning Approach

Our approach for using hybrid-memory is applicable to in-memory DBMS which store their data in a column oriented fashion. As a starting point we concentrate on the architecture of SAP HANA.

SAP HANA is offering two different states for data, *hot* and *cold*. Hot data is referring to columns that are loaded into main memory and can be accessed by the DBMS without additional overhead of reading data from disk whereas cold data is not loaded into main memory but is stored in the disk-based persistence layer. Cold data is loaded into main memory on demand when accessed or can be loaded directly by the user or application using Structured Query Language (SQL). As described in [11] tables will be unloaded from memory only when the system is running out of memory, or can by unloaded manually by using SQL. The potential transitions of loading and unloading data are shown in Figure 2. This figure also visualizes our extension which enables the use of hybrid-memory. In SAP HANA it is possible to set or alter an unload priority between zero and nine for a whole table using SQL. This way tables having a higher unload priority will be evicted from memory first.

The idea of the load and unload approach is to make all frequently accessed columns available in memory ensuring low query response times and to store any unused tables or columns on the HDD-based persistence layer. For very large datasets this approach involves certain drawbacks. Whenever a cold column is requested and the system is running out of memory, SAP HANA unloads whole tables in order to free up as much memory as needed for columns requested. Afterwards the requested columns have to be fully loaded

into DRAM, generating an overhead. We have evaluated this behavior for various table sizes as shown in Figure 3. The analysis makes it obvious that query performance will be critically degraded in case a DRAM limit is reached. Today *Scale-Out* would be the only option avoiding this effect.

Particularly for systems in which the database size exceeds available DRAM it can be advantageous to offer a more fine grained load and unload mechanism of data instead, especially because many applications work on a minor subset of the stored information, only. In these cases frequently loading and unloading full tables is no option to keep the system running when memory limits of the server are exceeded and Scale-Out remains the only option involving the mentioned drawbacks. Using hybrid-memory can be an alternative for scaling systems by offering an additional memory-tier to DRAM which could be used for data in a special state which is defined between hot and cold, and is referred to as *warm* in the remainder of this paper.

We introduce hybrid-memory as an additional store for columns as this seems well suited for a columnar DBMS, as well as for wide tables because it can be assumed that not all attributes are accessed in the same frequency. In our approach we partition tables into hot, warm, and cold columns. A column is referred to as *warm* when it is accessed infrequently and therefore stored in a slower but larger memory type such as hybrid-memory. Columns are *hot* when they are frequently accessed and should therefore remain in DRAM. The partitioning of data is implemented by using a special allocator for warm columns while hot columns keep using the standard allocator for DRAM. Additionally, a horizontal partitioning of data for warm columns is implied by the architecture of hybrid-memory which is done by *paging*. Any data allocated on the hybrid-memory is organized in pages sized 4kB. These pages are read from SSD and evicted from main memory whenever space is needed. The resulting architecture is shown in Figure 4.

The architectural design of hybrid-memory influences the partitioning of data into hot and warm. As our approach for hybrid-memory relies on paging it is not well suited for sequential data access. As mentioned, columns allocated on hybrid-memory are loaded into and evicted from memory using chunks of data sized 4kB. The size of the DRAM backed buffer is limited to a small fraction of the actual size of the warm data store to make the major amount of main

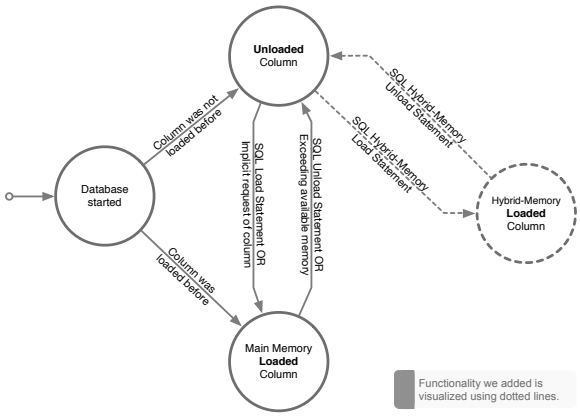


Figure 2: Process of implicitly and explicitly loading and unloading columns.

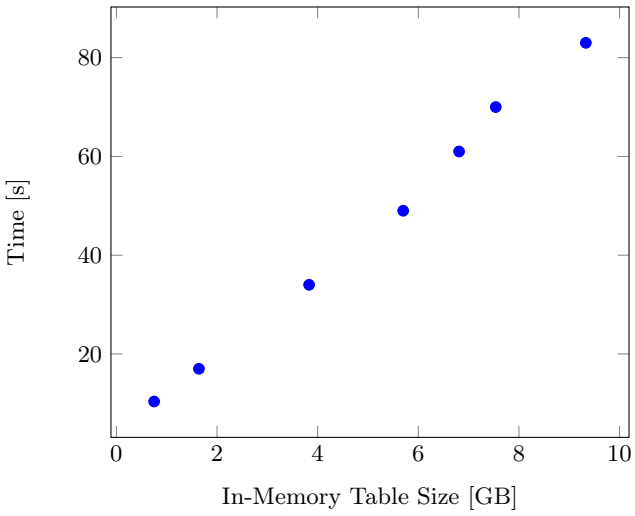


Figure 3: Database initial request times for unloaded tables related to their size in-memory.

memory available for hot data. Therefore, fully scanning warm columns would lead to a sequential load of all related pages into the buffer which therefore could cause the eviction of a number of other pages whenever the data size of warm columns exceeds the buffer size. We propose to partition data related to our vertical partitioning approach as outlined in Algorithm 1.

Algorithm 1 Determination of warm data

```
1: function dataWarm(columns, memoryLimit)
2:     //columns ordered ascending by usage frequency
3:     memory ← columns.sizeInMemory()
4:     warm ← []
5:     while memory > memoryLimit do
6:         column ← columns.next()
7:         cond ← column.isCondition()
8:         index ← column.isIndexed()
9:         if (NOT cond OR index) then
10:            memory -= column.size()
11:            warm.append(column)
12:        end if
13:    end while
14:    return warm
15: end function
```

In order to distinguish between hot and warm data, statistics of the access frequency of columns are needed. The access frequency describes how often a column is accessed compared with all queries within the time frame traced. Additional information defining if a column is used as a condition within a *Where* statement and information if it is indexed is needed. If a column is used in a condition without an existing index, a sequential scan needs to be performed on the column data which would be an expensive operation using hybrid-memory. Also, we do not want to introduce additional indexes in order to bypass the problem as it would not follow the principles of a columnar in-memory DBMS. Hence, hybrid-memory is mainly suited for reading small datasets avoiding scans of whole columns.

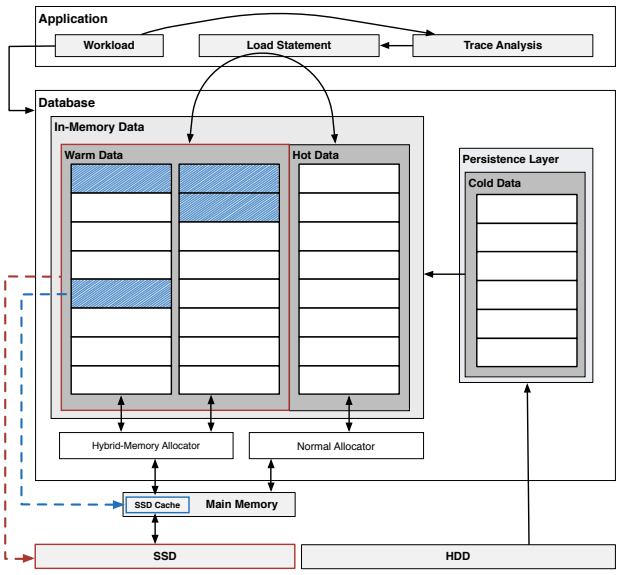


Figure 4: Integration of *warm* data into the existing states of hot and cold data.

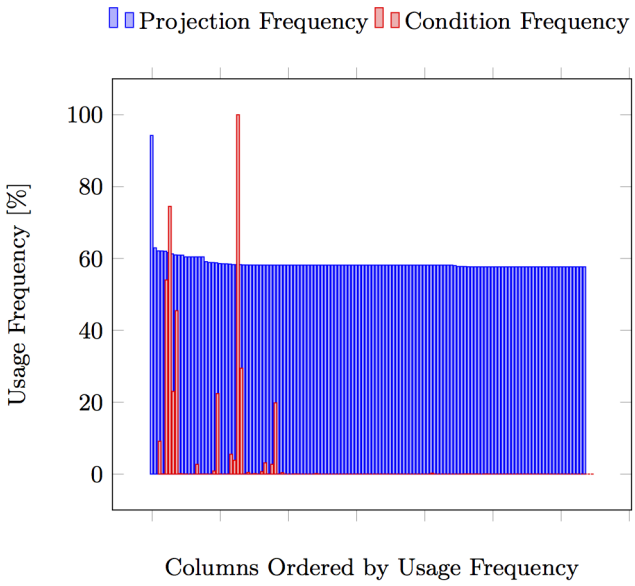


Figure 5: Projection and condition frequency for the columns of a single table.

Vertical data partitioning does not need to be re-run for existing and unchanged applications as the access patterns to data are defined by the application logic. Therefore analyzing a representative database trace of an existing application once would be sufficient to define columns to be loaded into hot and warm storage. An additional analysis is required only, whenever application access to data changes. We have not integrated data partitioning in the DBMS but implemented it in a separate preparation task instead. The result of the analysis is considered a fixed input of our system. By contrast, the main memory limitation can be adjusted at any time, which determines the size of the warm and hot data partition. The partitioning of data into hot and warm as shown in Figure 2 and 4 can be achieved in our DBMS by using the following SQL statements.

```
/* Load table or single columns into DRAM
or into hybrid-memory when WARM flag is
set */
LOAD <table> {ALL | (<column>, ...)} [WARM]
```

The SQL Data Manipulation Language (DML) has been extended adding the optional expression *"WARM"*. Whenever the DML load statement is executed by the user or application using the identifier *"WARM"* data will be allocated using hybrid-memory. For any other case the load and unload command is kept unchanged. The definition of cold data remains unchanged and simply refers to data which is never used and therefore not loaded into memory at all.

2.2 Data Access Analysis

We analyzed our approach using a database trace of a productively used ERP system. The trace represents a number of samples taken from the DBMS during the system's regular operations and consists of 1,444,163 queries in total, 975,757 of which are *Select* statements. We concentrate on vertical partitioning as no information about query results was available.

Our investigation concentrates on the following questions:

- How are projections and conditions spread among columns of a table?
- Is there an access frequency skew between all tables?
- What does the total access frequency for all columns look like?

First we look into the usage frequency of the columns of one table separated into projections and conditions. Figure 5 shows those results for a selected table. Looking at the projection frequency shows that almost 60% of all queries are accessing all columns. Hence, a high number of *Select ** statements is performed on the table and only 40% of all queries work on a subset of all columns. Such a case is unsuitable for vertically partitioning data, as no clear distinction regarding the access frequency for the columns of such a table is possible. However, the performance impact might not be that severe if the size of all accessed database tuples would not exceed the hybrid-memory buffer size and the same database tuples get accessed over and over again so that no cache misses occur. Continuing with the discussion of column partitioning we have found that most of the columns accessed in the *Where* statement have an index which means they are not scanned sequentially. Hence, they can be stored on hybrid-memory as well.

Another approach for partitioning might be derived from the general access skew of the tables in the DBMS. We have traced the access to all tables as shown in Figure 6 and all columns as shown in Figure 7. Out of 20 tables only 5 tables are accessed in more than 6% of all queries. By implementing a partitioning on table granularity we could save 65% of DRAM by moving tables which are accessed in less than 6% of the queries to hybrid-memory. Those tables are only used in 20% of all queries in total. Therefore, even in a worst-case scenario the performance of 80% of all queries would not be affected.

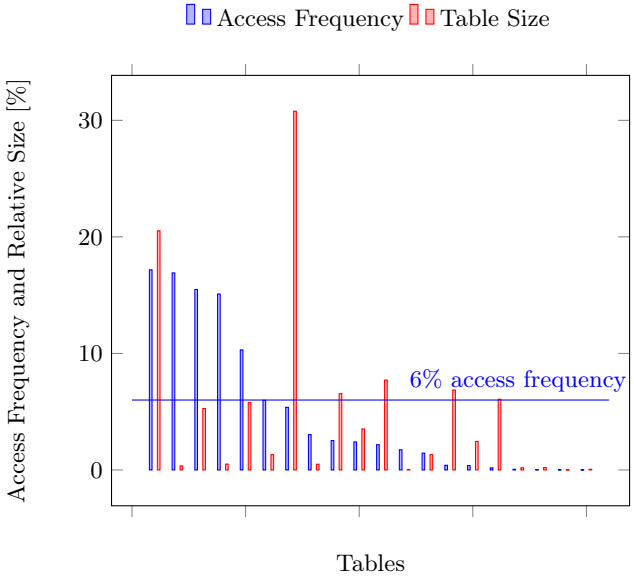


Figure 6: Table access frequency and sizes.

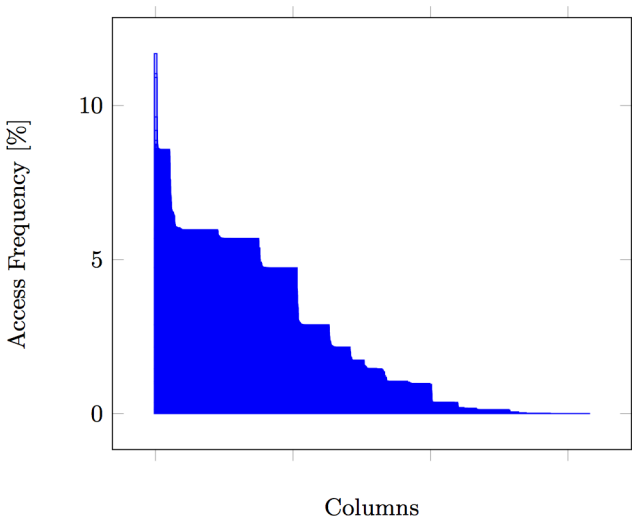


Figure 7: Column access frequency.

Finally, we combine both analysis by generating an access frequency chart for all columns taking the overall access frequency into account. Only about one-third of the columns are accessed in more than 5% of all queries. A potential vertical partitioning can leverage this information and use it to prioritize columns to be stored as warm data whenever the DBMS is running out of memory.

A full analysis of the available trace would have only been possible by taking a look into the actual rows returned by the DBMS. This would have allowed an analysis of the suitability of horizontal partitioning. Unfortunately, this was not possible. We are going to reiterate this approach in Section 4.1.2 when looking into the TPC-C workload.

3. HYBRID-MEMORY

This section provides an overview of hybrid-memory and how it is made available to the DBMS. Additionally, we will present a first evaluation of the technology using a microbenchmark simulating DBMS data access behavior.

3.1 Integration

When looking at the latencies of the memory hierarchy shown in Table 1 the huge difference between persistent storage and memory becomes obvious. SCM's latency and bandwidth are in the same order of magnitude of DRAM with a much higher capacity. However, as the capacity of SCM is limited by the number of available DIMM slots and it is still significantly slower than DRAM it might neither replace HDD nor DRAM. Hence, it is assumable that SCM will take its place as a third available storage technology within servers.

Besides emulating the technology a number of hardware vendors are working on bridge-technologies to overcome the time until SCM will be available. Those technologies as developed by Fujitsu Technology Solutions GmbH combine flash drives with a DRAM buffer. This technology is named hybrid-memory. It can be allocated like DRAM and is accessed byte-wise. As soon as SCM becomes available the hybrid-memory libraries can be replaced transparently to use it instead. At the moment hybrid-memory works similar to the concept of *paging* as known from Linux. In case the amount of virtual memory used by the application exceeds the available physical memory size, the software based mapping layer requires to reclaim pages from DRAM. This way selected pages are swapped to SSD and the required pages on SSD are re-transferred to the freed space in DRAM and thus become available in main memory again. This mechanism is extended by additionally offering the limitation of the available DRAM buffer size, introducing a predefined read-ahead functionality, as well as early page eviction. Hence hybrid-memory extends the available amount of main memory as shown in Figure 8.

L1 Cache	≈4 cycles
L2 Cache	≈10 cycles
L3 Cache	≈40–75 cycles
L3 Cache via QPI	≈100–300 cycles
RAM	≈60 ns
RAM via QPI	≈100 ns
SCM	≈80 – 10 000 ns
SSD	≈80 000 ns
HDD	≈5 000 000 ns

Table 1: Latencies of the memory hierarchy.

In contrast to the swapping algorithm of the Linux Operating System (OS), hybrid-memory can be accessed based on user needs. This way it becomes an additional memory-tier next to DRAM increasing the application's influence on preserving memory performance. Hence, we have integrated hybrid-memory as a separate data store into the columnar in-memory DBMS, SAP HANA. We refer to this data store as warm data which can be used additionally to the DRAM based hot data store. This way the actual functionality of the columnar in-memory DBMS stays unchanged. Paging is offered as an additional concept extending the available

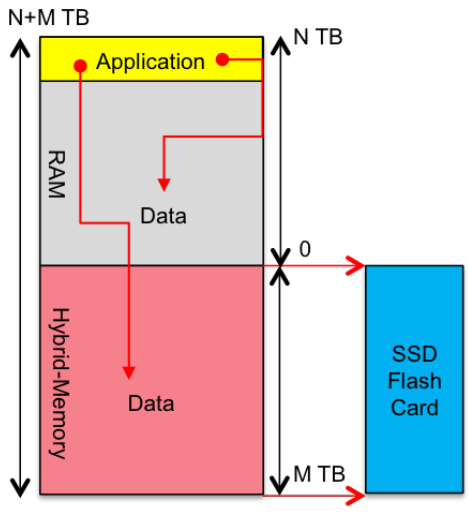


Figure 8: Hybrid-Memory integration into system.

amount of memory in the DBMS with no need for scaling the database across several servers. HANA stores data in two different stores, main and delta. The delta store is optimized for insert and update operations whereas the main store is read-optimized and read-only. Insert and update operations are performed on the delta store which is merged into the main store asynchronously. Additionally, all data is compressed using dictionary compression.

We integrated hybrid-memory into the main store only. Therefore mainly read operations are performed on hybrid-memory. All internal columns, indexes, and the delta store are always stored on DRAM. For the main store it can be chosen whether data is stored on hybrid-memory or DRAM.

3.2 Initial Benchmark

Hybrid-memory has been initially evaluated by using typical data access patterns of a DBMS. We used a microbenchmark simulating sequential scans as well as random reads on a single column being represented by a vector. This vector has been filled using 64-bit random integer values and can by be partitioned using different memory allocators for each partition. Hence, partitions allocated on DRAM as well as on hybrid-memory can be created. We ran two different scenarios using the microbenchmark:

1. No data partitioning is used. The entire vector gets allocated using either DRAM or hybrid-memory.
2. Data is partitioned into 35% of hot data allocated on DRAM and 65% of warm data allocated on hybrid-memory. Data access is skewed accessing hot data in 80% and warm data in 20% of all cases.

For each of those configuration we performed sequential scan and random read benchmarks. The vector size was set to 5 GB in all cases. The benchmark results show the effect of a limited physical memory in the system or offered to the hybrid-memory. Hence, paging is needed in order to make all data available. All configurations use a fast Peripheral Component Interconnect Express (PCIe) SSD as a backing store for DRAM. We used four measuring points limiting the memory between 100% and 40% of the total size of the vector.

In case data is not partioned performance results of the standard Linux paging algorithm and the implementation of the hybrid-memory are close to equal. Hybrid-memory shows a minor improvement in case of sequential data access. This does not apply for random access because prefetching can not be applied.

In case of data partitioning and skewed data access the effect of using a slower storage medium is weakened as shown in Figure 9. Because only 20% of the access addresses hybrid-memory, the impact of higher latencies is reduced.

Concluding, the microbenchmark offers first insights into the effect of using a slower memory tier for sequential scan and random read operations. However it simplifies from running a real DBMS. We take a look onto the integration of our approach into a modified development version of SAP HANA in Section 4.

4. EVALUATION

We evaluate the approach of hybrid-memory for scaling columnar in-memory DBMS focusing on the impact of hybrid-memory towards database performance.

Our evaluation focuses on two major problems:

- Reducing the amount of main memory consumption of the in-memory DBMS by vertical partitioning.
- Sizing the hybrid-memory buffer in order to minimize performance degradation of introducing data partitioning.

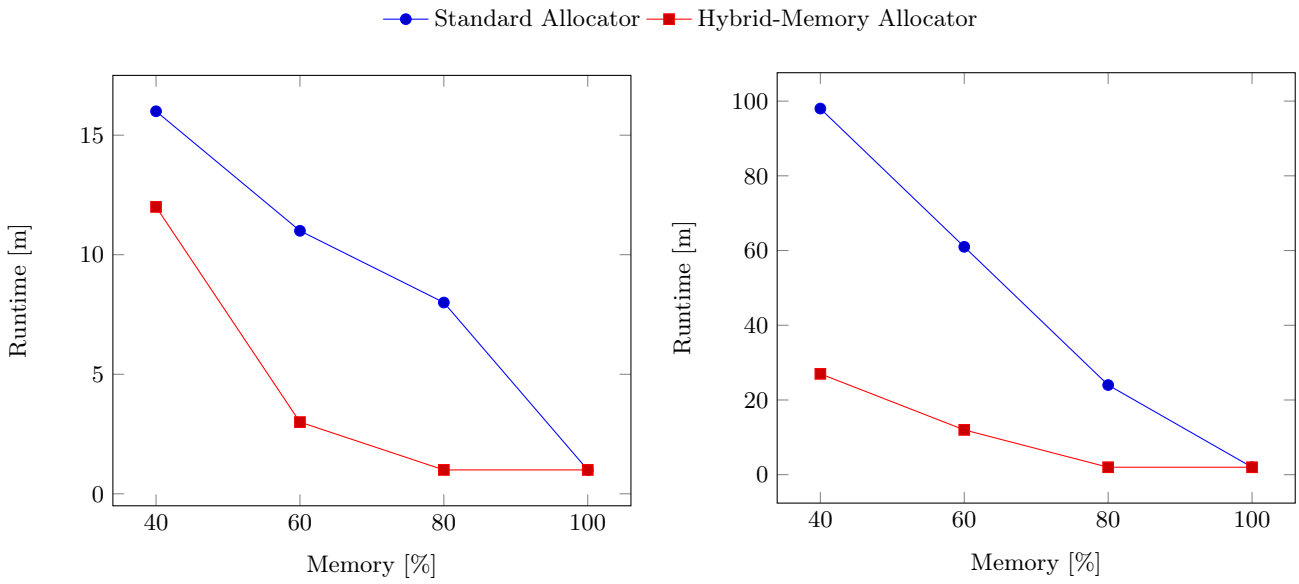
We will answer those questions for the TPC-C benchmark. In the following we start by introducing TPC-C, giving a short introduction into its workload and how it can be vertically partitioned. Next we present our TPC-C based evaluation results.

4.1 TPC-C Benchmark

The TPC-C benchmark is a commonly known OLTP DBMS benchmark approved by the Transaction Processing Performance Council (TPC) in 1992 [19]. Its standard specification can be obtained from [30]. Its workload describes the processes of a wholesale supplier having a number of sales districts and associated warehouses. Customers are placing new orders or requesting status existing orders where information is received by the DBMS. In the following we describe the workload in more detail focusing on its applicability for our approach. Afterwards we introduce vertical partitioning in order to make TPC-C utilising the hot and warm data store.

4.1.1 Workload

The TPC-C workload focuses on a DBMS which is used by the employees of a wholesale supplier executing transactions from their terminals in order to enter and deliver orders, record payments, check the status of orders, and to monitor the current stock of the warehouses. It consists of five different transaction types, namely new order, payment, order status, delivery, and stock level. A minimum execution frequency for each transaction type is specified by the TPC-C standard besides from the new order transaction. The amount of new order transactions measured is defined as the main metric of TPC-C named *tpmC*. It describes the number of new order transactions finished by the DBMS per



(a) Sequential Read **(b) Random Read**

Figure 9: Comparison of allocator performances using a microbenchmark simulating sequential and random column access. Data is partitioned into one-third of hot and two-thirds of warm data. 80% of the operations are performed on hot data, 20% on warm data.

minute which reflects the business throughput of the wholesale supplier. The dataset of the benchmark consists of nine tables having a total amount of 92 columns. It can be scaled defining the number of warehouses within the system. Accordingly all column sizes will be adopted except for the *item* table that reflects the products sold by the wholesale supplier.

Configuration	Setting
warehouses	50
virtual users	50
iterations	10000

Table 2: TPC-C configuration used for evaluation.

Our implementation of TPC-C relies on the open source database load testing and benchmark tool *Hammerora* [13]. Hammerora offers multiple parameters which can be changed by the user, e.g., the number of *warehouses*, *virtual users*, and *execution iterations*. The number of *warehouses* scales the dataset of the benchmark. *Virtual users* is used to set the maximum number of emulated, simultaneously working terminal operators executing transactions against the DBMS. *Iterations* defines the number of total transactions executed per user. Therefore it controls the runtime of the benchmark. Alternatively a timed run can be performed while iterations are executed until the defined time limit is reached. For the following evaluation these settings will be defined as stated in Table 2.

While using TPC-C we focus on the impact of introducing hybrid-memory to the DBMS. As Hammerora does not implement a full specification of TPC-C's metrics, we use transactions-per-minute (tpm) as the main metric comparing performance results of different DBMS configurations. As we are not focusing on actual DBMS throughput but rather on potential performance degradation we will present our results as variations from an unchanged DBMS. The following chapter concentrates on the vertical partitioning of TPC-C in order to make it applicable for our partitioning approach.

4.1.2 *Dataset Partitioning*

This paragraph concentrates on adopting TPC-C's dataset in order to make it applicable for vertical partitioning based on its workload. We will analyze the dataset and workload to answer the following questions:

- Is the workload of TPC-C suited to outline effects of introducing hybrid-memory?
- How can we partition TPC-C vertically to reduce the latency overhead of hybrid-memory?
- Is TPC-C workload beneficial for paging introduced by hybrid-memory?

For the configuration given in Table 2 we have analyzed the resulting database sizes.

	Initial		Final	
	Main	Delta	Main	Delta
Data	544 MB	2 MB	544 MB	330 MB
Dictionary	2571 MB	4 MB	2571 MB	1824 MB
Indexes	279 MB	0 MB	279 MB	377 MB
Sum	3394 MB	6 MB	3394 MB	2531 MB
Total	3400 MB		5925 MB	

Table 3: Initial and final TPC-C data, dictionary and index sizes distributed on main and delta store of SAP HANA.

The aggregated table sizes shown in Table 3 describe the dataset distribution of the TPC-C dataset in the database before and after a complete run of the benchmark. Data has been merged into the main store before the start of the benchmark. After the execution the dataset has grown by a total amount of 74%. The delta merge is disable in the DBMS in order to preserve constant performance of the database. The high data growth is caused by a nearly equal number of *Select* and *Insert / Update* statements within TPC-C's workload. Based on this we need to further analyze if TPC-C is mainly working on newly inserted or updated tuples as hybrid-memory is introduced into the main data store only. Effects of hybrid-memory only become visible whenever read operations access existing and unchanged data. Therefore we analyzed the number of *Select* statements accessing the main store and the delta store. The results listed in Table 4 show that most of TPC-C's read transactions are performed on the main store. Therefore TPC-C is potentially suited to make the performance impact of introducing hybrid-memory visible as three-fourth of the *Select* statements are accessing the main store.

Table	Main	Delta
Customer	28%	5%
District	0%	4%
Item	19%	0%
New Order	2%	0%
Orders	2%	0%
History	0%	0%
Order Line	1%	19%
Stock	15%	3%
Warehouse	0%	2%
Total	67%	33%

Table 4: Distribution of data access to main and delta store grouped by tables in relation to all *Select* statements during a single TPC-C run.

Next we analyze how TPC-C can be vertically partitioned while reducing the overhead of introducing hybrid-memory as good as possible. Our analysis focuses on *Select* statements first as they are accessing the hybrid-memory. We run TPC-C in order to obtain its workload using the *SQL Plan Cache* of the DBMS. Afterwards we parse the plan cache using a simple SQL grammar building an abstract syntax tree. Finale we traverse the tree counting the usage of each column. We separate column usage into sequential and random access. Sequential access of columns refers to scan operations generated by the *Where* statements of a query used to find corresponding rows. Columns used for projections rather generate a random access to memory as only subsets of the data are used during the materialization of a result depending on the *Where* condition of the query. We found that for TPC-C between one and ten rows are materialized for each *Select* statement. This separation is necessary as hybrid-memory can cope with random access whereas sequential access of columns potentially causes significant overhead. Based on this knowledge we finally create a Least Frequently Used (LFU) ordering of all columns. Columns scanned sequentially are ranked highest priority to stay in main memory if they are not indexed. Columns accessed randomly are prioritized according to their usage frequency. The process of analyzing the dataset is shown in Algorithm 2.

Algorithm 2 Generating LFU list for all columns

```
 1: function columnsLFU
 2:     lfu ← []
 3:     runWorkload("TPCC")
 4:     planCache ←fetchDBPlanCache("TPCC")
 5:     for all query in planCache do
 6:         ast ←parsePlanCache(query)
 7:         traverseAST(ast,&lfu,query.count)
 8:     end for
 9:     lfu ← sortMergeLFU(lfu)
10:     return lfu
11: end function
12:
13: function traverseAST(ast,&lfu,count)
14:     if seqAccess(ast.current()) then
15:         if hasIndex(ast.current()) then
16:             lfu.appendRndAcc(ast.current(),count)
17:         else
18:             lfu.appendSeqAcc(ast.current(),count)
19:         end if
20:     else if rndAccess(ast.current()) then
21:         lfu.appendRndAcc(ast.current(),count)
22:     end if
23:     for all child in ast.getChildren() do
24:         traverseAST(child,&lfu,count)
25:     end for
26: end function
```

Based on the LFU the dataset can be vertically partitioned. We found that only 15 columns of TPC-C are accessed in more than one-sixth of all *Select* statements. Overall more than 30% of all columns are projected in less than 4% of all *Select* statements whereas even 40% of all columns are not projected at all. The final share of data in the hot and warm store is set based on memory limitations of the server. Storing as much frequently used data as possible on DRAM using the LFU preserves the overall performance of the DBMS.

Finally hybrid-memory also implies a horizontal partitioning based on its nature of paging chunks of data. Whenever rows are accessed which are already stored in the DRAM based cache of hybrid-memory no disk access is needed. If applications constantly access the same data the overhead of using hybrid-memory is reduced. We have analyzed TPC-C workload towards its capabilities to leverage paging. We instrumented the stored procedures of TPC-C storing the row identifiers of all *Select* statements. Next, we simulated the access of TPC-C to the *Customer* table and assumed a limited amount of cache. As shown in Figure 10 hybrid-memory does not need to access the underlying SSD when the buffer size is not smaller than 60%. The number of page faults increases rapidly for lower buffer sizes.

Taking a look onto the *Update* statements of TPC-C we found that updates of the database are done identifying the corresponding rows using indexes. Therefore no further analysis needs to be done towards those statements, as indexes are not stored on hybrid-memory.

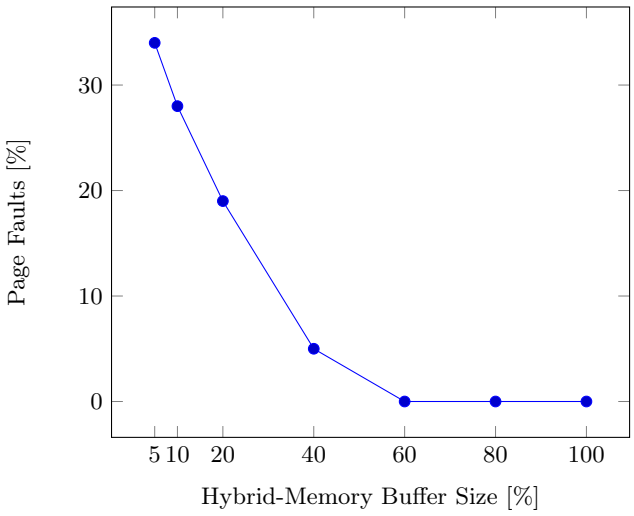


Figure 10: Approximated number of page faults for *Customer* table running TPC-C workload. *Customer* table is assumed to be fully stored on hybrid-memory only adjusting the buffer size.

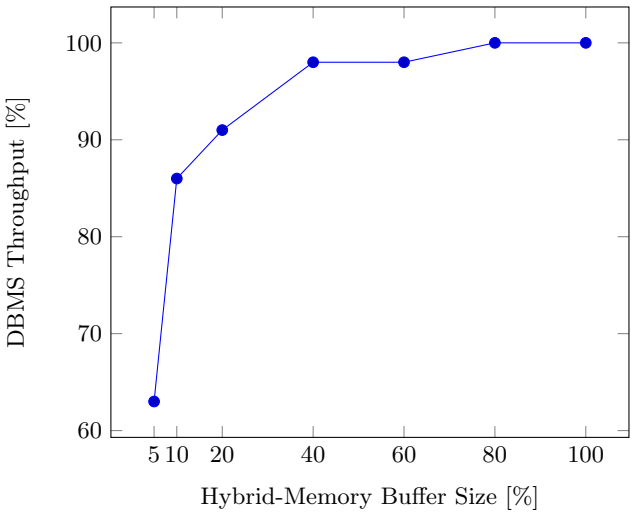


Figure 11: DBMS performance in percent of tpm compared to unchanged DBMS for different hybrid-memory buffer sizes and a constant share of hot and warm data of 20% by 80%.

4.1.3 Test System

For our evaluation we used a Fujitsu Primergy RX600 S6 equipped with four Intel Xeon CPU E7-4870, 512GB of DRAM, a 768GB Intel P3700 SSD used as hybrid-memory backing store and 512GB of HDD. The system was running Suse Enterprise Server 11 SP2.

4.2 Results

For the TPC-C benchmark we assumed a fixed vertical partitioning of data into hot and warm. In total about 20% of all columns were stored on DRAM which were the most frequently accessed columns. The remaining columns have been allocated using hybrid-memory. Hence, the amount of DRAM directly used by the attribute vectors could be reduced by 60%.

We adjusted the size of the DRAM based buffer of hybrid-memory analysing its impact on query throughput and optimizing the amount of main memory saved. We found that the DBMS throughput is degraded less than 15% if the size of the hybrid-memory buffer is reduced by not more than 10% . Hence, the total amount of DRAM used by attribute vectors of the modified SAP HANA was reduced by 54%. This enables a potential scaling of the attribute vectors by more than 100% with no need for a Scale-Out accepting just a minor performance degradation.

In conclusion scaling SAP HANA based on hybrid-memory offers a good solution to reduce the amount of DRAM needed while still achieving high query throughput performance.

5. RELATED WORK

Novel storage technologies like SCM have been studied by various researchers in the past. Work concentrated on two different aspects of using SCM, scaling and data persistence.

Ramos et al. [24] present an approach on scaling servers based on SCM using a hardware-driven page placement efficiently reducing the amount of DRAM in the system. Dhiman et al. [5] present a hybrid hardware-software solution to manage the combination of SCM and DRAM. A software based suggestion for integrating SCM is done by Qureshi et al. [22] reducing system power and cost. In contrast to our approach SCM is seen as a replacement for DRAM reducing the amount of it to a minimum. Contrary we propose SCM to become an additional memory layer.

Persistence characteristics of SCM, which we do not address in this paper, as they could for example be used for logging are proposed by Oukid et. al [20] for in-memory DBMS. The transition of data from disk to SCM in disk-based DBMS is presented by Tavares et al. [29] and Wang et al. [33]. Application independent usage of the persistence of SCM is shown by Condit et al. [3] and Venkataraman et al. [31].

There are a number of different papers that address the capacity limitation of in-memory DBMS. In H-Store an approach called *Anti-Caching* is used to tackle the problem [4]. In contrast to the traditional approach of caching they do not try to keep active data in memory, but instead make sure that archived data is pushed to disk. A comparable technique is used in Hekaton [6]. Here, cold data is also pushed to secondary storage, in this case flash memory. In their work the authors describe how the system can transparently access a tuple without knowledge about its location. In Hyper a method called *Compacting* is used [10]. Similar to our approach the data stored in a columnar layout is partitioned horizontally and each partition is categorized by its access frequency. Data in the (rarely accessed) *frozen* category is still kept in *RAM* but compressed and stored in huge pages to save memory. Finally, an approach for identifying hot and cold data in an in-memory DBMS is presented by Levandoski et al. [27]. Contrary to our work data is either stored in memory or on disk. An additional store is not assumed.

6. CONCLUSION AND OUTLOOK

In this paper, we presented an approach for hybrid-memory scaling columnar in-memory databases. We investigated how to introduce hybrid-memory into a columnar in-memory

DBMS by vertically partitioning data into hot and warm columns. Additionally, we implemented our approach into a development version of SAP HANA in order to evaluate and optimize the reduction of DRAM consumption. We found that main memory used by the attribute vectors can be reduced by more than 50% while decreasing the DBMS performance less than 15%.

We believe that our approach can also be applied to SCM which would potentially further reduce the performance overhead. For further work, we plan to integrate hybrid-memory into additional parts of SAP HANA, supporting a partitioned data dictionary.

7. REFERENCES

[1] D. P. Bovet and M. Cesati. *Understanding the Linux kernel*. O'Reilly, Sebastopol, CA, 2006.

[2] G. W. Burr, B. N. Kurdi, J. C. Scott, et al. Overview of candidate device technologies for storage-class memory. *IBM*, 52(4.5):449–464, July 2008.

[3] J. Condit, E. B. Nightingale, C. Frost, et al. *Better I/O Through Byte-Addressable, Persistent Memory*. 2009.

[4] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, Sept. 2013.

[5] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th DAC*, DAC '09, page 664–469, New York, NY, USA, 2009. ACM.

[6] A. Eldawy, J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. In *PVLDB Vol. 7, Issue. 11, June 2014*. VLDB – Very Large Data Bases, Sept. 2014.

[7] R. Elmasri. *Fundamentals of database systems*. Addison-Wesley, Boston, 6th ed edition, 2011.

[8] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, July 2008.

[9] D. Fried, C. Skinner, and J. Loesche. HP and SAP advance SAP HANA through joint innovation, May 2013.

[10] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP OLAP databases. *CoRR*, abs/1208.0224, 2012.

[11] F. Färber, N. May, W. Lehner, et al. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[12] M. Grund, J. Krüger, H. Plattner, et al. HYRISE: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, Nov. 2010.

[13] HAMMERDB. HammerDB, 2014.

[14] B. Höppner, O. Lilienthal, H. Schmitz, et al. SAP HANA in a hybrid main memory environment. In *HPI Future SOC Lab: proceedings 2013*. Universitätsverlag Potsdam, Apr. 2013.

[15] A. Kemper and T. Neumann. HyPer: A hybrid OLTP OLAP main memory database system based on virtual memory snapshots. pages 195–206. IEEE, Apr. 2011.

[16] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle TimesTen: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[17] P.-a. Larson, M. Zwilling, and K. Farlee. *The Hekaton Memory-Optimized OLTP Engine*.

[18] J. Lee, Y. S. Kwon, F. Farber, et al. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. *2013 IEEE 29th ICDE*, 0:1165–1173, 2013.

[19] S. T. Leutenegger and D. Dias. A modeling study of the TPC-c benchmark. *SIGMOD Rec.*, 22(2):22–31, June 1993.

[20] I. Oukid, D. Booss, W. Lehner, et al. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. Snowbird, USA, June 2014.

[21] H. Plattner. *In-memory data management: an inflection point for enterprise applications*. Springer, Heidelberg, 2012.

[22] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. ISCA '09, page 24–33, New York, NY, USA, 2009. ACM.

[23] V. Raman, G. Attaluri, R. Barber, et al. DB2 with BLU acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, Aug. 2013.

[24] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the ICS*, ICS '11, page 85–95, New York, NY, USA, 2011. ACM.

[25] SGI. SGI to preview in-memory appliance at SAPPHIRE® NOW, June 2014.

[26] V. Sikka, F. Färber, W. Lehner, et al. Efficient transaction processing in SAP HANA database: The end of a column store myth. SIGMOD '12, page 731–742, New York, NY, USA, 2012. ACM.

[27] R. Stoica, J. J. Levandoski, and P.-A. Larson. Identifying hot and cold data in main-memory databases. ICDE '13, page 26–37, Washington, DC, USA, 2013. IEEE Computer Society.

[28] M. Stonebraker, V. Inc, A. Weisberg, and V. Inc. *The VoltDB Main Memory DBMS*.

[29] J. A. Tavares, J. d. A. M. Filho, A. Brayner, and E. Lustosa. SCM-BP: An intelligent buffer management mechanism for database in storage class memory. *JIDM*, 4(3):374–389, 2013.

[30] Transaction Processing Performance Council (TPC). TPC BENCHMARK c standard specification revision 5.11, Feb. 2012.

[31] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX*, FAST'11, page 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[32] T. Vogelsang. Understanding the energy consumption of dynamic random access memories. pages 363–374. IEEE, Dec. 2010.

[33] A.-i. A. Wang, P. Reiher, and G. J. Popek. Conquest: better performance through a disk/persistent-RAM hybrid file system. In *In Proceedings of the 2002 USENIX*, page 15–28, 2002.