Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory

Robert Lasch
TU Ilmenau, SAP SE
Walldorf, Germany
robert.lasch@tu-ilmenau.de

Thomas Legler
SAP SE
Walldorf, Germany
thomas.legler@sap.com

Norman May
SAP SE
Walldorf, Germany
norman.may@sap.com

Bernhard Scheirle
SAP SE
Walldorf, Germany
bernhard.scheirle@sap.com

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

ABSTRACT

The cost of DRAM contributes significantly to the operating costs of in-memory database management systems (IMDBMS). Persistent memory (PMEM) is an alternative type of byte-addressable memory that offers — in addition to persistence — higher capacities than DRAM at a lower price with the disadvantage of increased latencies and reduced bandwidth. This paper evaluates PMEM as a cheaper alternative to DRAM for storing table base data, which can make up a significant fraction of an IMDBMS' total memory footprint. Using a prototype implementation in the SAP HANA IMDBMS, we find that placing all table data in PMEM can reduce query performance in analytical benchmarks by more than a factor of two, while transactional workloads are less affected. To quantify the performance impact of placing individual data structures in PMEM, we propose a cost model based on a lightweight workload characterization. Using this model, we show how to place data pareto–optimally in the heterogeneous memory. Our evaluation demonstrates the accuracy of the model and shows that it is possible to place more than 75 % of table data in PMEM while keeping performance within 10 % of the DRAM baseline for two analytical benchmarks.

PVLDB Reference Format:
Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory. PVLDB, 15(11): 2867 - 2880, 2022.
doi:10.14778/3551793.3551837

1 INTRODUCTION

In-memory database management systems (IMDBMS) store table data in DRAM, achieving superior performance over traditional disk-based systems. However, data volumes managed by IMDBMS are continuing to grow while the costs and capacities of DRAM are stagnating. This makes pure in-memory systems cost-ineffective and has sparked research into data systems that do not rely on DRAM exclusively but still approach in-memory performance [24, 36, 43] or that try to optimize the cost-performance tradeoff by placing data across a range of heterogeneous storage devices [47].

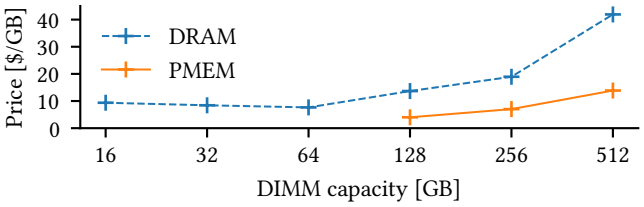

Figure 1: DRAM and PMEM prices (August 2020) [35].

A promising hardware technology in the in-memory context is persistent memory (PMEM), a new type of byte-addressable memory, which has become commercially available in recent years. PMEM is a cheaper alternative to DRAM at higher capacities per DIMM, as illustrated by Figure 1, which shows DRAM and PMEM prices per GB from August 2020 [35]. Higher capacities per DIMM are desirable for memory-intensive workloads, as the number of DIMMs that can be attached to a single CPU is limited. Besides memory used for intermediate results, the main memory footprint of an IMDBMS is significantly driven by table data. As PMEM is byte-addressable, IMDBMS table data can be placed into it without significant code changes. By placing table data in PMEM, it should thus be possible to lower the memory cost of operating IMDBMS and to enable managing larger data volumes on a single node. Placing table data in *persistent* memory can also reduce restart times as data does not necessarily need to be reloaded from disk on restart, like it is the case with volatile memory [2, 37]. However, the downside of PMEM compared to DRAM is lower performance, namely roughly 3× higher read latencies and 3× lower maximum read bandwidths, and even lower write bandwidths [22].

In this paper, we investigate the performance impact of placing IMDBMS table data in PMEM to answer the following questions:

- How much is query execution slowed down if table data is placed in PMEM instead of DRAM?
- Can the slowdown for a given workload be predicted based on statistics collected with minimal overhead?
- How should data be distributed in DRAM and PMEM to trade off runtime performance and memory cost?

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551837

To answer these questions, we conduct an experimental analysis of table data placement in PMEM and DRAM using SAP HANA [15], a commercial IMDBMS. To decide which structures to place in PMEM with minimal impact on query performance, we have already proposed a heuristic based on lightweight access counters collected during workload execution in previous work [23]. However, this only yields a single point on the cost-performance tradeoff which is also not guaranteed to be pareto–optimal. We address this here by proposing a cost model for data placement in PMEM that can be used for pareto–optimal placement decisions.

After introducing our experimental setup in Section 2, the paper is split into three main parts, each addressing one of the abovementioned questions: Section 3 analyzes the slowdown incurred by placing data in PMEM for both analytical and transactional workloads. For this, we use the analytical TPC-H and TPC-DS benchmarks, and OLTP-like point queries, on two hardware systems using two versions of the currently available PMEM modules. The insights from this section serve as the basis for Section 4, where we propose a cost model for table data placed in PMEM. The model uses microbenchmarks of access primitives in combination with access counters collected during workload execution to predict the performance impact of placing individual data structures into PMEM. Using the model, we show how to make pareto–optimal placement decisions with a configurable cost–performance tradeoff in Section 5. Our evaluation using TPC-H and TPC-DS shows that 75 % of the referenced table data can be placed in PMEM with little performance impact. We also compare to PMEM's hardware-managed Memory mode. In Sections 6 and 7 we discuss related work and conclude.

2 BACKGROUND

Below, we discuss relevant background information on PMEM and column stores, as well as our experimental setup.

2.1 Persistent Memory

In the past, several persistent memory (PMEM) technologies have been proposed. So far, only Intel Optane DC Persistent Memory Modules (DCPMM) based on 3D-XPoint technology are commercially available [19, 20]. PMEM offers byte-addressability and direct persistence at (nearly) DRAM speed, as well as a higher density and better economic characteristics than DRAM as shown in Figure 1.

DCPMM provides two possible operating modes: Memory and App Direct mode. Memory mode allows applications to use PMEM as larger volatile memory, where DRAM acts just as a kind of L4 cache and, therefore, no rewrite of in-memory software is necessary. In order to utilize the persistency property of PMEM and to control data placement in PMEM or DRAM at the application level, App Direct mode must be used, where DCPMM appears as a separate memory device. We thus focus on App Direct mode. The read latency for DCPMM is 2–3× higher than for DRAM [22]. Also, the bandwidth of DCPMM is lower than DRAM bandwidth, i.e., for sequential reads up to 3×, for random reads higher.

2.2 Domain-Encoded Column Stores

SAP HANA splits columnar table data into *main* and *delta* storage, with most of the data residing in the read-only main storage, and delta storage regularly being merged into main storage [15]. The delta storage is only used for data modifications, and thus columns which are not changed store 100 % of data in main storage. As the delta storage is typically an order of magnitude smaller than main storage, and subject to latency-critical write operations, it is best kept in DRAM. In this study, we thus focus on the main storage as the largest memory consumer in the average SAP HANA system. The main storage uses order-preserving dictionary compression (also called domain encoding) to compress data and accelerate certain operations. There are three major data structures for each column in main storage: *dictionaries*, *data vectors* (also called index vectors), and *inverted indices*, of which dictionaries and data vectors are the core structures. The *dictionary* is a mapping from *value identifiers* to actual values. Dictionaries are always sorted by value. The *data vector* stores a *value identifier* for each row in that column, i.e., it maps *row identifiers* to *value identifiers* (cf. [26, Fig. 1a]). This method results in a reduction of the data volume for columns with repeated values. Data vectors can use any of several available compression methods to further reduce the memory footprint [26]. For dictionaries, specialized implementations exist for fixed-size types like integers, and variable-size types like strings and large objects (LOBs). String dictionaries are additionally compressed using prefix encoding [44]. The *inverted index* is an optional third structure that maps each value identifier back to a set of row identifiers, i.e., it performs the inverse mapping of the data vector. It can be used for example to speed up column scans. As inverted indices can have a substantial memory footprint they are typically only maintained for a subset of all columns, e.g., primary key columns. We refer to [1] and [39] for further details on domain-encoded column stores.

2.3 Experimental Setup

Table 1: Hardware systems used in this paper.

System	**100 Series**	**200 Series**
Microarchitecture	Cascade Lake	Icelake
CPU Model	Xeon Platinum 8260L	Xeon Platinum 8368
Cores/Threads	24/48	38/76
L2 Cache (per Core)	1024 KiB	1280 KiB
L3 Cache	36 608 KiB	58 368 KiB
DRAM (per Socket)	6 × 16 GiB	8 × 16 GiB
DRAM Spec.	DDR4-2666	DDR4-3200
PMEM (per Socket)	6 × 256 GiB	8 × 512 GiB
PMEM Type	Optane PM 100 Series	Optane PM 200 Series
Operating System	SLES 12 SP4	SLES 15 SP2

Hardware Configuration. We use two systems, one with Intel® Optane™ **100 Series** Persistent Memory, and one with the newer Intel® Optane™ **200 Series** Persistent Memory. 200 Series Optane is advertised to feature slightly higher bandwidths than 100 Series Optane [19, 20]. System details are listed in Table 1. Both systems have two sockets, but we limit experiments to a single socket to avoid NUMA effects, unless noted otherwise. Each PMEM DIMM shares a CPU memory channel with one DRAM DIMM. Thereby using only DRAM reaches the same bandwidth that would be reached with only DRAM in the system [21]. The PMEM in both systems is configured in App Direct mode. We measured idle latencies of 99 ns and 322 ns for DRAM and PMEM on the **100 Series** system, and

84 ns and 298 ns on the **200 Series** system for random 64-byte reads. The maximum read bandwidths are 9.8 GiB/s and 16.6 GiB/s on 100 and 200 Series PMEM, respectively. Note that sequential reads can reach up to four times higher bandwidths, as they can fully utilize the internal 256 B access granularity of the PMEM DIMMs [48].

Software Configuration. Although SAP HANA already supports placement of entire columns in PMEM [2], we implement the placement of individual data structures in PMEM by extending SAP HANA's buffer cache [43], using the built-in PMEM block manager (cf. [2]). We do not use the Persistent Memory Development Kit (PMDK). See [23] for more details. This solution allows us to decide for each structure (data vector, dictionary, inverted index) per column whether it is placed in DRAM or PMEM. All experiments are conducted using this prototype which is otherwise fully equivalent to the SAP HANA Cloud Edition from Q4 2020. We size the buffer cache sufficiently large such that no data is evicted and that all data used by queries is preloaded into the cache before running queries. Note that this setup implies that never-accessed columns will not be loaded into memory. All queries are executed using the HANA Execution Engine (HEX), SAP HANA's state-of-the-art code-generating query execution engine.

3 PLACING TABLE DATA IN PMEM

To establish a performance baseline for main storage data placed in PMEM instead of DRAM, we benchmark analytical queries as well as small read-only queries that would be typical for transactional workloads. We compare the case where all data is placed in DRAM to the case where all main storage structures are placed in PMEM on both of the hardware systems introduced in Section 2.3.

3.1 OLAP

To gather an understanding of OLAP workloads on PMEM, we consider the TPC-H and TPC-DS benchmarks. Both benchmarks are run at scale factor (SF) 100. For the datasets, we use default compression settings and indices on the primary keys of all tables. We measure each of the queries defined by the benchmark individually, as well as a consecutive run of the queries. We always include a warm-up run followed by five consecutive runs of the respective query or queries to obtain stable results. In these initial experiments, we compare the case where all table data is placed in PMEM to the baseline scenario where all data is placed in DRAM. In addition to comparing the **100 Series** and **200 Series** systems, we also include results from the **200 Series** system where we use PMEM from the *far* NUMA node instead of NUMA-local PMEM. While the direct comparability between **100 Series** and **200 Series** is limited due to the different CPU models, cache sizes, and DRAM specifications, the far-NUMA experiment allows us to directly compare the slowdowns incurred by PMEM with different latencies on an otherwise equal hardware system. We observed additional latencies of 65-75 ns for a NUMA hop, which is similar to the latency difference between Optane 100 Series and Optane 200 Series DIMMs. It is also expected that upcoming disaggregated memory technologies such as CXL's `.mem` protocol will exhibit additional latencies of around 100 ns [40]. The experiment can thereby also serve as a comparison point for data placed on possible future disaggregated memory.

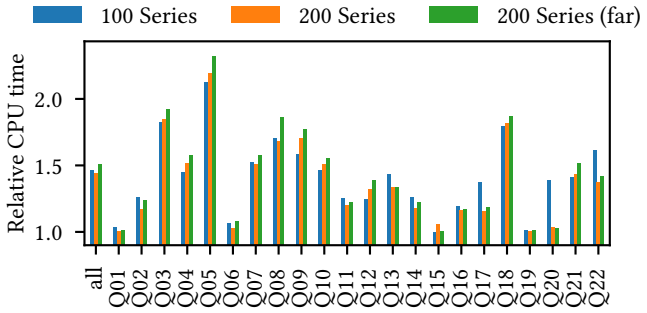

Figure 2: TPC-H (SF 100) CPU times when all data is placed in PMEM relative to all data placed in DRAM.

TPC-H. Figure 2 shows the CPU time needed to execute a full run of all 22 TPC-H queries combined sequentially or each query individually relative to the CPU time when data is placed in DRAM. We show (relative) CPU times here and for the remainder of the paper, as they carry equivalent information to the actual query response times. We have observed equivalent slowdowns in query response times, as the amount of parallelization remains the same regardless of whether data is placed in DRAM or PMEM.

On the **100 Series** system, the TPC-H queries experience significant slowdowns of up to 120 % when data is placed in PMEM as opposed to DRAM. A full consecutive run of all queries is 46 % slower on the **100 Series** system and 44 % slower on the **200 Series** system, while the median slowdown across all queries is 40 % and 33 %, respectively. On the newer **200 Series** system the slowdowns are generally less severe, due to the lower latency of Optane 200 Series DIMMs in comparison to Optane 100 Series. This is not always the case however, due to the differences in core count and CPU cache sizes across the systems, which also significantly speed up the DRAM baseline on the **200 Series** system in comparison to the **100 Series** system. The faster DRAM baseline can cause a higher slowdown on the **200 Series** system. However, when we compare the difference in CPU time between the two memory types on both systems, the difference on the **200 Series** system is always lower than on the **100 Series** system. Using far-NUMA instead of local PMEM on the **200 Series** system consistently results in higher slowdowns than using the local PMEM. The only exceptions are Q15 and Q20, which are short-running queries that experience little slowdown, making them subject to measurement noise. With far-NUMA PMEM, the full run of queries slows down by 51 %, while the median slowdown across all queries is 36 %. It is also notable that the increase in slowdown from NUMA-local to far-NUMA PMEM is not uniform across all queries. This suggests that the sensitivity to higher memory latency varies across the queries.

TPC-DS. For TPC-DS we observe smaller slowdowns than for TPC-H. We omit TPC-DS Q38 as it could not be run on the **100 Series** system due to insufficient DRAM capacity. Across the remaining 98 TPC-DS queries, the median slowdown is 5.7 % on the **100 Series** system and 3.8 % or 4.4 % on the **200 Series** system (for local or far-NUMA PMEM), with worst-case slowdowns of 40 % and 39 % or 63 %, respectively. The reason for the comparably smaller effect of PMEM–placement on TPC-DS compared to TPC-H is the

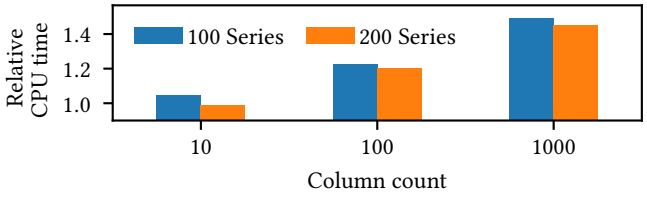

Figure 3: OLTP SELECT query on INTEGER table with 1M rows and varying column counts. CPU time is relative to CPU time with data placed in DRAM.

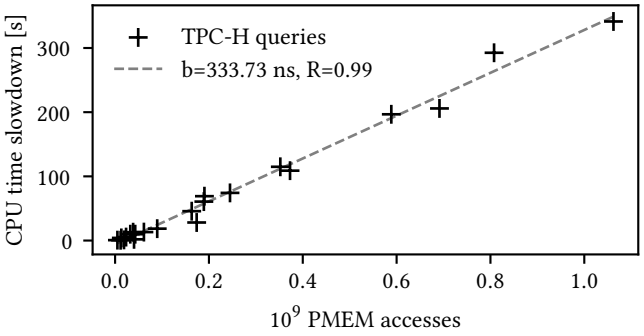

Figure 4: Cache misses for TPC-H queries.

general higher complexity of the TPC-DS queries. This causes less time to be spent accessing base data in relation to the time spent on processing intermediate results. Thus, when we place base data in PMEM, the slowdown is less noticeable compared to TPC-H.

3.2 OLTP

To gain insights into the performance of OLTP workloads, we use a point query of the form `SELECT(*) FROM table WHERE key=X`, where `key` is unique. `table` has a varying number of integer columns and an inverted index on `key`. This allows us to measure the impact of PMEM placement on tuple materialization. Note that we do not consider a more complete OLTP workload that also includes writes, as any writes would be handled by the delta store in SAP HANA, which remains in DRAM in this study. We therefore only focus on the read-only parts of OLTP here.

Figure 3 shows how much the `SELECT` query is slowed down depending on the number of columns in the table. As with the OLAP benchmarks, the reported value is the CPU time spent on executing the query when table data is placed in PMEM relative to the CPU time spent when table data is placed in DRAM. With only ten columns, there is little slowdown of the query. This is because the additional latency of tens of PMEM accesses needed for tuple materialization instead of DRAM accesses is much lower than other overhead in processing the query such as parsing and planning. This changes with more materialized columns, where roughly 20 % and 40 % slowdown can be observed for 100- and 1000-column tables, respectively. Because of the lower latencies of the 200 Series Optane, the slowdowns on that system are always slightly lower in comparison to the **100 Series** system. Prior work investigating OLTP workloads on PMEM has also already shown that the incurred slowdown only becomes apparent if tuples need to be materialized from hundreds or thousands of columns [41].

Because we limit the scope of this paper to investigate placement of *main storage* in PMEM, we focus further evaluation on OLAP workloads. Placement of the main storage in PMEM does not adversely affect the DML queries that make up significant fractions of OLTP workloads, as those would be handled by delta storage, and we have just shown the minor impact data placement in PMEM has on the point queries which are common in OLTP workloads. For the rest of the paper, we also focus on the newer **200 Series** system, as our investigation has shown that reporting results for both systems would be redundant.

3.3 PMEM Cache Miss Latency

The reason for the slowed down query execution when data is placed in PMEM can be traced back to the higher latencies of PMEM in comparison to DRAM. Specifically, any access to a data structure placed in PMEM that does not hit the CPU caches experiences the higher latency of PMEM as opposed to the lower DRAM latency it would experience if the structure was placed in DRAM.

To illustrate this, we use hardware counters to measure the total number of CPU cache misses that are serviced by PMEM for the execution of each TPC-H query: In Figure 4, each point represents one query, with the number of CPU cache misses that were serviced by PMEM shown on the x-axis, and the difference in CPU times between executing the query with all data in DRAM and with the main storage in PMEM shown on the y-axis. The figure shows that the measured slowdown directly correlates with the number of PMEM accesses. A linear regression reveals the average CPU time cost *per access*. The regression line has a slope of 334 ns, which suggests that the average PMEM access incurs an additional latency of 334 ns in comparison to a DRAM access. That latency difference is consistent with the difference in loaded latency between DRAM and PMEM we have measured in preliminary measurements at a bandwidth utilization of around 5 GiB/s, which is also the average bandwidth utilization per query during the benchmark. Higher latency differences could occur if the PMEM read bandwidth was saturated further during the benchmarks, but when measuring bandwidth utilization, we have not observed queries that fully saturate the PMEM bandwidth for extended periods of time. We believe that this is a somewhat special property of compressed column stores like SAP HANA's main store. Combining dictionary compression and additional lightweight compression techniques results in significantly smaller data volumes that need to be accessed to retrieve the same data, while increasing the amount of computation necessary to decode the data, making it much less likely run into bandwidth limitations even when executing large scans using all available cores. To illustrate, we ran TPC-H Q06, the most scan-heavy query in TPC-H [14], in MonetDB, which stores columns as flat arrays in memory — i.e., uncompressed. While MonetDB and SAP HANA took nearly the same time to execute the query using all available cores on the **200 Series** system, MonetDB fully utilized the available DRAM bandwidth for the entire query duration, while HANA's peak bandwidth utilization remained below 15 GiB/s.

4 MODELLING SLOWDOWNS

Next, we construct a model for the slowdowns observed in Section 3. The goal is to provide estimates of the increase in CPU time needed to process a set of queries when any given set of column store data structures is placed in PMEM instead of DRAM. The expected benefits of such a model are twofold:

First, one should be able to use the model as a means of judging the impact of placing a known DRAM-resident workload fully or partially in PMEM. The model can facilitate this by providing an estimate of the PMEM–induced performance impact for the workload, which can be used for targeted decision-making based also on the cost-savings of switching from a DRAM-only to a hybrid DRAM–PMEM memory system. Second, in such a hybrid memory system, it should also be possible to make *optimal* decisions for data placement based on the model. This can be achieved by using the model to predict the performance impact of PMEM–placement for each data structure accessed by the workload individually. In combination with the structure's memory footprint and a given performance or DRAM budget, pareto–optimal placement decisions can be made. We evaluate model–driven placement in Section 5.

For the model, characterization of the workload and memory system is necessary, which we discuss in Sections 4.1 and 4.2. We construct and evaluate the model in Sections 4.3 and 4.4.

4.1 Workload Characterization

We have seen in Section 3.3 that the slowdown incurred by data placement in PMEM is proportional to the number of memory accesses that would have been served by DRAM in a DRAM-only setup, but are instead served by PMEM in the case where the data is placed in PMEM. Thus, to model the slowdown, we need to be able to predict the number of memory accesses to PMEM for a given workload. To do this, we employ lightweight *access counters* implemented within the DBMS, with the goal of characterizing the workload independently of the memory subsystem used at the time of recording the counters. These counters track the *total number of logical accesses* per data structure (data vector, dictionary, inverted index) of each column in the dataset during workload execution. Counting accesses per data structure allows us to use the resulting model to make placement decisions at data structure granularity. Because the data structures may be accessed in different ways that can result in varying numbers of physical accesses to memory, we differentiate between different *types* of accesses. In combination with microbenchmarks that measure the cost of individual accesses on both types of memory (cf. Section 4.2), this later allows us to model the PMEM–induced slowdown.

The two basic types of accesses we count are *lookups* and *scans*. A lookup accesses a single entry in a data structure based on an identifier, while a scan accesses multiple or all entries with the goal of finding particular entries matching a search predicate. In terms of the memory access pattern, a lookup constitutes, except for the case discussed below, one or multiple random memory accesses, while a scan results in sequential accesses to memory. We count each accessed entry of the structure as one *access*, i.e., a full scan over a data vector with 1M entries would be counted as 1M *scan accesses*. In practice, scans are only performed on data vectors, for mapping value identifiers to row identifiers, and on dictionaries

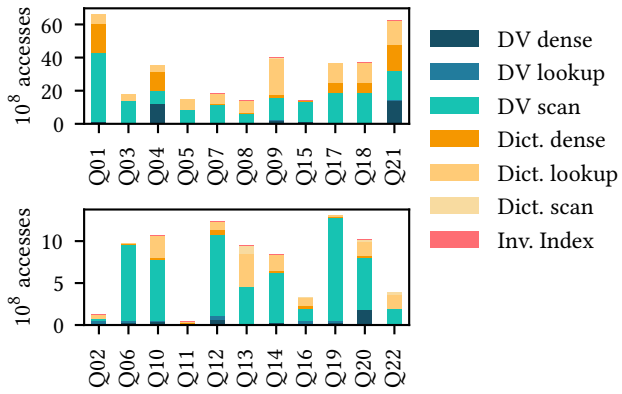

Figure 5: Aggregated access counters for TPC-H.

for string searches with `LIKE` predicates. As dictionaries are sorted, other operations that need to map values back to value identifiers can use binary searches, where the accesses are lookups [39].

We further differentiate lookups into fully random and *dense* lookups. This is based on the observation that, e.g., when materializing tuples, it is often the case that many nearby entries of the same data structure are looked up sequentially. Like scans, dense lookups result in sequential memory access patterns, which can be predicted by hardware prefetchers and thus induce lower additional costs than fully random lookups when data is placed on PMEM. Dense lookups can occur in varying degrees of locality, i.e., with different *strides* between the nearby entries that are looked up, with higher memory costs at higher strides. As the difference in access latency between DRAM and PMEM for these lookups varies based on the stride, we split dense lookups into three classes based on the stride. This is discussed in more detail in Section 4.2.

While more advanced approaches for workload characterization exist [8] that maintain more accurate statistics at block-level, e.g., with the goal of advising physical database design, we are only interested in quantifying the impact of PMEM–placement per data structure and thus only count the total number of accesses per structure. Because of this, the performance and space overhead of the proposed counters is negligible. In our experiments, we obtain the access counter values for a given set of queries while running baseline experiments with all data placed in DRAM. In practice, counter collection could also be done online as the workload evolves, and is independent of whether data is placed in DRAM or PMEM. The updated values obtained in this fashion could be used to periodically revisit model–driven placement decisions. The same applies for completely unknown workloads. Here all data could be initially placed into PMEM. Then, after collecting access counters for a few minutes to get a representative sample of the workload, model–driven placement could be used to place parts of the data in DRAM, e.g., based on the available DRAM capacity.

As an example, Figure 5 shows the collected access counters for each individual TPC-H query, aggregated across all columns in the dataset. In the figure, the three classes of dense lookups are grouped together for data vectors (DV) and dictionaries (Dict.), and accesses on inverted indices are grouped into a single category due to their

low overall prevalence. The figure shows how the total accesses as well as the composition of the access types vary across the queries. Of special interest is the high prevalence of dense data vector (Q04, Q20, Q21) and dictionary (Q01, Q04, Q21) lookups in certain queries. This affirms our decision to count such dense lookups separately. We also show in Section 4.4 how conflating dense lookups with random lookups would affect the model's accuracy.

To fully characterize the workload, additional metadata is needed for each column: First, the *data type* needs to be known, as access costs may differ between data types. Second, the data vector *compression method* must be known, as some compression methods introduce indirections for each access [26] and thus also affect access costs. Third, the *size* of each structure in memory needs to be known to make placement decisions which reduce the memory footprint as much as possible. All of this data can be obtained from the DBMS's monitoring views when running the workload for characterization or directly from the database schema.

4.2 Memory System Characterization

As a second step towards the cost model, we need a characterization of the involved memory types that tells us the expected slowdown for each different *type* of access tracked by the access counters when a structure is placed in PMEM instead of DRAM.

To obtain this characterization, we perform microbenchmarks of the different access types described in Section 4.1, scans, lookups, and dense lookups. The microbenchmarks measure the access latencies on both DRAM and PMEM. The latency differences between the memory types represent the slowdown of any particular access type when a structure is placed in PMEM instead of DRAM and the access does not hit the CPU caches. An alternative to running microbenchmarks would be to directly use measurements of the memory access latencies. However, this would require analyzing the implementation of each type of access for each structure, potentially also for different compression methods and data types, to be able to model cache miss behavior. Thus, microbenchmarking the different accesses on both DRAM and PMEM to obtain a black-box model of the access costs is more suitable.

We use the Google benchmark library [16] for the microbenchmarks. In addition to the different access types, for dictionaries we also differentiate between integer and string dictionaries. For data vectors we perform microbenchmarks for each of the five compression methods supported by SAP HANA, as well as for uncompressed data vectors. We generate columns with 128M unique random values for the benchmarks, which results in a data vector, dictionary, and inverted index with 128M entries each. For string dictionary benchmarks we generate random strings with 10 characters. The microbenchmark performs the corresponding access on the respective structure using a single thread until the benchmark library has determined a stable result for the access latency. For lookup accesses we make sure to chain the accesses in a way that makes the location of each lookup depend on the result of the previous one. This prevents the CPU's out-of-order execution from running multiple lookups in parallel and skewing the results.

Lookups and scans. A subset of the microbenchmark results is shown in Table 2. The table shows the latencies for random lookups and — where applicable — scans on the three data structures and on

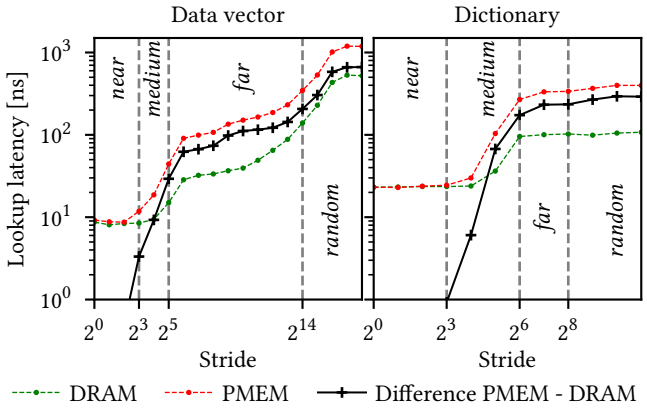

Figure 6: Dense lookup microbenchmark results.

both evaluated hardware systems. Besides the absolute latencies for data placed in DRAM and PMEM, the difference between PMEM and DRAM latency (Diff.) is also shown, as this is the metric later used in our cost model. Dense lookups are discussed separately in more detail below. Comparing the **100 Series** to the **200 Series** results in the table shows that the latter system performs better both for DRAM and for PMEM accesses. This is not surprising, as the 200 Series Optane DIMMs are advertised to feature slightly lower latencies compared to the 100 Series Optane DIMMs, and as the DRAM latency on the **200 Series** system is also slightly lower. Looking at the latency differences between PMEM and DRAM, there is a large contrast between random lookups and scans.

For scans, due to their sequential memory access pattern and because each loaded cacheline can contain multiple structure entries, the difference between DRAM and PMEM is in the range of a single nanosecond per scanned structure entry. Note that the difference even becomes negative for string dictionary scans on the **100 Series** system. This is because measurement noise resulted in a slightly higher value for the DRAM latency than for the PMEM latency here. For use in the model we clamp negative values to zero.

For lookups, there are significant differences in latency up into the microsecond range, as each lookup can incur several random memory accesses, which have higher latencies on PMEM than on DRAM. This highlights the importance of treating lookups and scans separately in the model. From the five data vector compression methods, we show only run-length-encoding (RLE) in the table as the most extreme example. Lookup latencies on RLE compressed vectors are several times higher than on uncompressed data vectors. In SAP HANA, RLE compressed data vectors replace sequences of the same value identifier with a single instance of the value identifier and its *start position*. Thus, to perform lookups, a binary search on the start positions has to be done, resulting in significantly more random memory accesses than a lookup on an uncompressed data vector. Other compression methods similarly show higher latencies than those on uncompressed data vectors.

Dense lookups. To characterize lookups accessing nearby structure entries, we microbenchmark *dense lookups* with strides between accessed structure entries including powers of two from 1 to

Table 2: Access microbenchmark results for random lookups and scans.

		100 Series			200 Series		
Structure	Access Type	DRAM	PMEM	Diff.	DRAM	PMEM	Diff.
Data Vector (uncompressed)	Lookup	691.4 ns	1304.4 ns	613.0 ns	589.4 ns	815.2 ns	225.8 ns
	Scan	0.5 ns	1.2 ns	0.7 ns	0.6 ns	1.0 ns	0.4 ns
Data Vector (RLE)	Lookup	2268.0 ns	4670.0 ns	2401.9 ns	1907.4 ns	3126.1 ns	1218.7 ns
	Scan	3.0 ns	4.4 ns	1.4 ns	3.2 ns	4.4 ns	1.2 ns
Dictionary	INT Lookup	146.7 ns	413.6 ns	267.0 ns	107.9 ns	372.5 ns	264.5 ns
	STRING Lookup	523.2 ns	860.1 ns	336.9 ns	483.8 ns	820.4 ns	336.6 ns
	STRING Scan	231.5 ns	231.2 ns	−0.3 ns	216.2 ns	216.2 ns	0.0 ns
Inverted Index	Lookup	680.6 ns	1035.0 ns	354.4 ns	602.8 ns	868.5 ns	265.7 ns

2^{18}. Figure 6 shows the lookup latencies for DRAM– and PMEM–resident structures on the **100 Series** system, as well as the DRAM–PMEM latency difference. We exemplarily show results for uncompressed data vectors and for integer dictionaries, but we observe qualitatively similar results leading to the same conclusions for compressed data vectors and string dictionaries. We observe distinctive *regions* that are characterized by the magnitude of the difference in DRAM and PMEM latency, and marked in the figure by vertical dashed lines. This leads to the classification into three classes of dense lookups that was already mentioned in Section 4.1. Strides lower than eight structure entries result in latency differences of less than a nanosecond. We call accesses in this region *near dense* lookups. These lookups often target the same cache line as the prior lookup and are thus predominantly L1 cache hits, resulting in a near-complete elision of the latency difference between DRAM and PMEM. The next region features a latency difference in the order of magnitude of ten nanoseconds, and we call those accesses *medium dense* lookups. Here the sequential nature of the access pattern is likely still detected by hardware prefetchers and accesses thus often result in L2 cache hits, partly hiding the latency difference. Accesses in the third region are called *far dense* lookups and are characterized by a latency difference in the order of magnitude of a hundred nanoseconds. Here each lookup incurs one or multiple cache misses, resulting in a more noticeable latency difference. Beyond this, the absolute latencies and latency differences approach the values shown for random lookups in Table 2, so we consider those lookups as random ones. When weighing dense lookups in the cost model, we use the access latency difference measured at the upper bound stride of the respective region to obtain a more conservative prediction of the slowdown. With our goal of using the model for placement decisions, this is reasonable: Underpredicting the slowdown incurred by a structure and placing it in PMEM — resulting in a higher slowdown than expected — would be worse than keeping the structure in DRAM. Lastly, we have not observed significant numbers of dense lookups for inverted indices. For those we thus only count directly adjacent lookups as dense, in favor of simplifying the access counting implementation.

CPU caches. The microbenchmark results represent the case where logical accesses result in cache misses and thereby physical memory accesses. This is of course not the case for all logical accesses, as memory requests can often be served by the CPU caches and thereby avoid a full-latency access to DRAM or — in our case — PMEM. As part of our characterization of the memory system we thus also consider the size of the L3 and L2 caches as important model parameters. More specifically, the model considers the sum of the L3 cache size and each core's L2 cache size as the "last level cache" size S_{LLC}. Starting from the Cascade Lake microarchitecture, the L3 cache on Intel CPUs is non-inclusive of the L2 cache, effectively making the total last level cache size the sum of both. All other relevant hardware properties are represented implicitly by the microbenchmark results.

4.3 Runtime Cost Modelling

The last important model input besides the workload and memory characterization is the *placement configuration* $P_{s,c}$. This describes for each data structure of each column in the dataset, whether it is placed in DRAM or PMEM:

$$P_{s,c} = \begin{cases} 1, & \text{if structure } s \text{ of column } c \text{ is placed in PMEM.} \\ 0, & \text{if structure } s \text{ of column } c \text{ is placed in DRAM.} \end{cases} \quad (1)$$

where $c \in C$ is one of the dataset's columns, C is the set of all columns in the dataset, and $s \in \{\text{v}, \text{d}, \text{i}\}$ is one of the three column store data structures, data **v**ector, **d**ictionary, or **i**ndex. With this, the total additional CPU time t needed to process a workload given a placement configuration $P_{s,c}$ is modelled as follows:

$$t = \sum_{c \in C} P_{\text{v},c} t_{\text{v},c} + P_{\text{d},c} t_{\text{d},c} + P_{\text{i},c} t_{\text{i},c} \quad (2)$$

where $t_{\text{v},c}$, $t_{\text{d},c}$, and $t_{\text{i},c}$ are the additional CPU times incurred *if the respective structure of column c is placed in PMEM.* $t_{\text{v},c}$, $t_{\text{d},c}$, and $t_{\text{i},c}$ are the core equations of the model. They weigh each access counter value by the respective additional cost of that type of access obtained from the microbenchmarks from Section 4.2:

$$t_{\text{v},c} = CMR(m_{\text{v},c}) \cdot \sum_{a \in A} x_{c,\text{v},a} \cdot b_{\text{v},a,cm_c} \quad (3)$$

$$t_{\text{d},c} = CMR(m_{\text{d},c}) \cdot \sum_{a \in A} x_{c,\text{d},a} \cdot b_{\text{d},a,dt_c} \quad (4)$$

$$t_{\text{i},c} = CMR(m_{\text{i},c}) \cdot \sum_{a \in A} x_{c,\text{i},a} \cdot b_{\text{i},a} \quad (5)$$

Here, $m_{s,c}$ is the memory footprint of the respective data structure s of column c and $a \in A$ is one of the five access types in $A = \{\text{scan}, \text{lookup}, \text{near dense}, \text{medium dense}, \text{far dense}\}$. $x_{c,s,a}$ is the access counter value for access type a on structure s of column c and b_{v,a,cm_c}, b_{d,a,dt_c}, and $b_{\text{i},a}$ are the additional CPU times needed for the access type a on the respective structure when that structure is placed in PMEM instead of DRAM, as measured in the

microbenchmarks. Note that for data vectors and dictionaries, the microbenchmark result also depends on the compression method cm_c or the data type dt_c of the column c, respectively.

Finally, the *cache miss rate* $CMR(m)$ models the effect of the CPU caches, using the memory footprint m of a data structure to estimate the fraction of accesses to the structure which result in *cache misses* and thereby contribute to the additional CPU time if the structure is placed in PMEM:

$$CMR(m) = \frac{1}{1 + e^{-(\frac{m}{S_{\text{LLC}}} + \alpha)}}, \text{ with } \alpha = \log \frac{1}{y_0} - 1 \tag{6}$$

We use a right-shifted sigmoid function to model the CMR, where S_{LLC} is the size of the last level cache, as discussed in Section 4.2. The offset α controls the y-intercept y_0, which we set to $y_0 = 5\,\%$. This choice of function is based on the observation that cache miss rates grow with the size of the working set until they reach 100 % once the working set is significantly larger than the cache size [13]. The function models this behavior by starting out at 5 % for small structures and gradually approaching 100 % as the structure size becomes a few multiples of S_{LLC}. The model makes the simplifying assumption that the cache miss rate of each structure is independent of the overall size of the working set. While more advanced modelling of the CMR with additional inputs would be possible [30] and probably more accurate, we find that this approach matches our real-world observations well enough and thus leave more advanced modelling as future work. Requiring additional inputs would likely also increase the complexity of the workload characterization, which we aim to keep lightweight. The model is agnostic of the number of used threads or CPU cores, as we model CPU times and assume that bandwidth saturation does not occur.

Limitations. The presented model is not without limitations and simplifying assumptions: First, it is assumed that the higher latency of PMEM accesses directly translates to an increase in CPU time. However, this is not necessarily the case, as the out-of-order processing of modern CPUs can exploit memory level parallelism of temporally nearby accesses, partly mitigating higher memory latency by interleaving multiple accesses. Our model makes the worst-case assumption that such access interleaving does not happen and that each access made to PMEM instead of DRAM incurs the full additional latency. This is sensible, as it results in more pessimistic estimates of the incurred slowdown. Prior work [41] has also shown that such latency hiding for PMEM requires targeted optimizations, which are not present in our case. Second, the model does not consider bandwidth limitations. While PMEM provides lower read bandwidths than DRAM, we did not encounter many instances of bandwidth saturation in our experiments — as already discussed in Section 3.3 — and thus omit this factor in favor of model simplicity. Modelling the effects of bandwidth saturation would require tracking accesses over time instead of for the entire workload duration, to be able to estimate bandwidth utilization over time. This would violate our initial design goal of keeping the workload characterization lightweight. Omitting this allows us to assume that the slowdown for any placement configuration is a linear combination of the slowdown incurred by each individual structure that is placed in PMEM, which also greatly simplifies the model–driven placement described in Section 5.

4.4 Model Accuracy

Using the measurements of PMEM-induced slowdowns on the **200 Series** system, we evaluate our cost model. We assess prediction accuracy by comparing measured to predicted slowdowns. An ablation study demonstrates the sensitivity of the model to the CPU caches and to handling dense lookups differently than random ones.

Prediction accuracy. We consider three placement configurations for the TPC-H and TPC-DS datasets:

(a) All data is placed in PMEM.
(b) Only data vectors are placed in PMEM, while dictionaries and inverted indices remain in DRAM.
(c) Only dictionaries are placed in PMEM, while data vectors and inverted indices remain in DRAM.

This allows us to not only assess the overall prediction accuracy of the model, but also its accuracy across the data structures used in the column store. Being able to make accurate predictions not only for PMEM–only placement but also for placing individual structures in PMEM is important for model–driven placement. We omit the case where only inverted indices are placed in PMEM as we never encountered slowdowns of more than 10 % in this scenario. For each placement configuration, Figure 7 compares the measured relative CPU time for executing the TPC-H and TPC-DS queries to the respective model predictions. The CPU times are relative to the case where data is placed exclusively in DRAM. Model predictions are obtained following Equation 2, with the shown relative value calculated as $\frac{T_{\text{DRAM}}+t}{T_{\text{DRAM}}}$, where t is the model prediction from Equation 2 and T_{DRAM} is the measured DRAM-only baseline CPU time for the respective query. Each point in the figure represents a single query. An *ideal* model would predict any measured slowdown perfectly, which would result in all points lying on the dashed gray line. As the calculation of the relative CPU time is sensitive to low query execution times due to measurement noise, we show queries which require less than 100 seconds CPU time with decreased opacity, marking them as less reliable results.

For PMEM–only placement (a) we generally measure less drastic slowdowns for TPC-DS queries than for TPC-H queries. No TPC-DS query experiences a slowdown of more than 50 %, while there are several TPC-H queries with higher slowdowns. Comparing the placement configurations shows that placing only data vectors in PMEM (b) has little effect on query performance. In contrast, placing dictionaries in PMEM (c) results in much larger slowdowns than the former configuration, which leads us to conclude that dictionaries placed in PMEM are also the main cause of slowdowns in the PMEM–only (a) configuration. This is due to the fact that accesses to dictionaries are more likely to be random lookups, whereas data vectors are mostly scanned, as the access counters for TPC-H in Figure 5 show. As Table 2 shows, random lookups on data structures placed in PMEM face a much larger penalty than scans.

Comparing the model predictions to the measured slowdowns, we find that the slowdowns larger than 50 % in Figure 7a and c are predicted well by the model. Although there are slight over- and underpredictions, the magnitude of the slowdown is predominantly predicted accurately. For the predictions of the smaller slowdowns, which can be seen more clearly in the versions of the plots that are zoomed into the 0 % to 30 % slowdown range, we can see that the accuracy of the predictions is slightly worse than for the higher

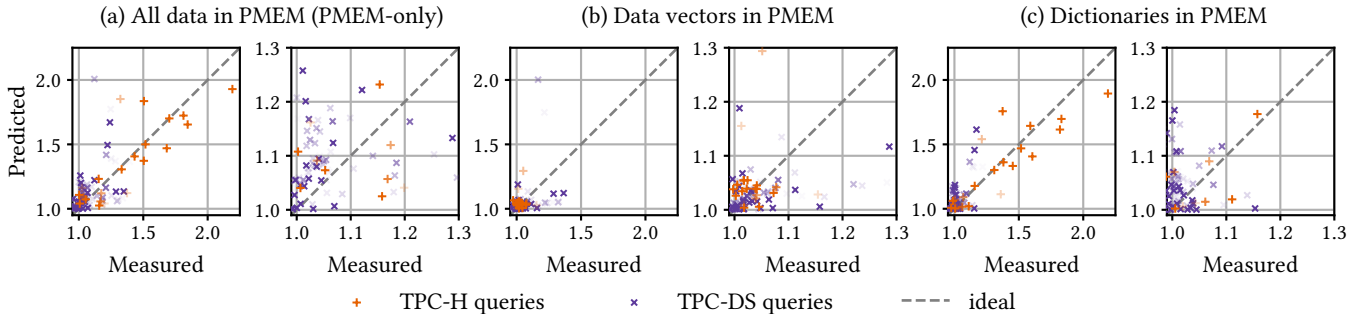

Figure 7: Comparison of predicted to measured CPU time relative to the DRAM-only configuration for three placement configurations. Queries with CPU time < 100 seconds are shown with decreased opacity. A zoom-in is shown for each configuration.

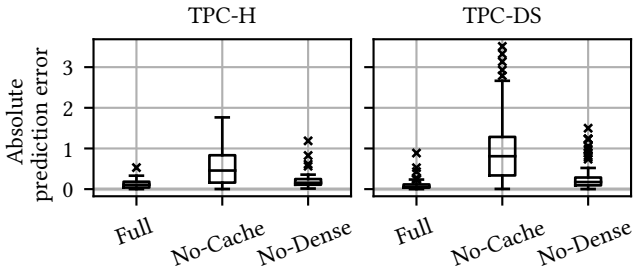

Figure 8: Sensitivity of the model to not considering CPU caching (No-Cache), and to treating dense lookups as random lookups (No-Dense), compared to the full model (Full).

slowdowns, i.e., there seem to be more predictions which deviate farther from the *ideal* line. However, the deviation from the *ideal* line still stays below 0.2 in most cases. We call this deviation the *absolute prediction error*, defined as $|(T_{\text{DRAM}} + t)/T_{\text{DRAM}} - T_P/T_{\text{DRAM}}|$. T_{DRAM} is the measured CPU time for DRAM-only placement and T_P is the measured CPU time for the respective placement configuration (a), (b), or (c), while t is the model prediction for the additional CPU time incurred by the placement. The worst-case absolute prediction error is 0.89 across the three placement configurations, while the median errors are 0.07, 0.02, and 0.04 for (a), (b), and (c), respectively. This shows that the model performs well across the different data structures used in the column store, and we argue that it is reasonably accurate given our goal of keeping the overhead of workload characterization as low as possible.

Ablation study. To confirm the importance of modelling the behavior of the CPU caches and that of separately counting dense lookups, we compare the predictions of the **full** model for PMEM–only placement to the predictions of two model variants:

(1) **No-Cache**: Ignore the CPU caches, i.e., Equation 6 is replaced by $CMR(m) = 1$, regardless of structure or cache size. This is equivalent to assuming that no CPU caches are present and each access to PMEM instead of DRAM incurs additional CPU time worth the full latency difference between DRAM and PMEM.
(2) **No-Dense**: Treat dense lookups as random lookups, i.e., all types of dense lookups are considered to be random lookups in Equations 3, 4, and 5.

For **No-Cache**, Figure 8 shows drastically increased median and peak prediction errors for both TPC-H and TPC-DS. While the median error of the full model is 0.11 and 0.06 for TPC-H and TPC-DS, respectively, it increases to 0.46 and 0.81 for the **No-Cache** variant. This demonstrates the importance of including the cache miss ratio into the model. Without this factor in the model, slowdowns are often overpredicted. The increased error is particularly prominent for TPC-DS. One reason for this is that the TPC-DS dataset mostly features columns with few unique values, which have dictionaries that fit into the CPU caches and thus experience a low cache miss rate for lookups. This is predicted adequately by the CMR factor in the full model. When leaving this factor out, the slowdown incurred by dictionaries in PMEM is vastly overpredicted. With the **No-Dense** variant the prediction errors also increase, to median values of 0.16 for TPC-H and 0.17 for TPC-DS. As with **No-Cache**, the increased errors are all due to overpredictions, caused here by using the full random lookup latency to weigh dense lookups. In contrast to **No-Cache** this does not negatively affect predictions for all evaluated queries, but it does severely affect a subset of the queries, as indicated by the increased maximum error values in comparison to the full model. The queries are not all equally affected by this change as not all queries perform a significant amount of dense lookups (cf. Figure 5).

5 MODEL–DRIVEN PLACEMENT DECISIONS

The model may also be used to make optimal placement decisions along the cost-performance trade-off, given sufficient accuracy. In practice, such placement decisions could be made in an offline process to make a static placement decision for a given workload, but it may also be desirable to adjust the placement decisions if the workload changes over longer periods of time. This is possible with our model, as workload characterization using access counters is lightweight and can be done online. In this section, we show how placement decisions can be made based on the proposed cost model and evaluate their results.

5.1 Budget– and Target–Based Placement

To use the cost model for optimal placement decisions, we formulate a mixed-integer linear program (MIP). If all model inputs are available this MIP problem can be solved using any MIP solver. We use the Python package `Python-MIP` [45] for this purpose. The

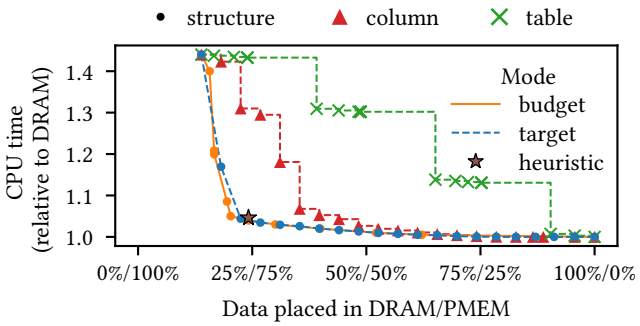

Figure 9: TPC-H placement mode/granularity comparison.

solution of the optimization problem is a placement configuration P, as defined in Section 4.3. Due to the simplicity of the cost model, the resulting optimization problem is essentially a 0-1 knapsack problem where the number of items is proportional to the number of columns in the dataset. This does not require long solve times — for single-threaded optimization using `Python-MIP`'s default `CBC` backend, we measure median optimization times of 19 ms and 69 ms for TPC-H and TPC-DS, respectively. We offer two placement modes:

The *budget mode* allows setting a performance budget b, while placing as much data as possible in PMEM. The performance budget b is defined as the maximum additional CPU time that may be used relative to the CPU time T_{DRAM} taken for the workload when all data is placed in DRAM. In this mode, the optimization objective is to maximize the data volume placed in PMEM

$$\sum_{c \in C} P_{\text{v},c} m_{\text{v},c} + P_{\text{d},c} m_{\text{d},c} + P_{\text{i},c} m_{\text{i},c} \text{ subject to } t \leq b \cdot T_{\text{DRAM}}$$

The *target mode* allows setting a memory target, which defines the fraction of the total data volume which shall be placed in PMEM. Here the optimization objective is to minimize t, the additional CPU time predicted by the model, subject to the constraint

$$\sum_{c \in C} P_{\text{v},c} m_{\text{v},c} + P_{\text{d},c} m_{\text{d},c} + P_{\text{i},c} m_{\text{i},c} \geq m_{target}$$

where m_{target} is the memory target in bytes. In our experiments, we use $target$, the minimum percentage of the total data volume to be placed in PMEM, to define m_{target}:

$$m_{target} = target \cdot \sum_{c \in C} m_{\text{v},c} + m_{\text{d},c} + m_{\text{i},c}$$

5.2 Placement Evaluation

We evaluate model–based placement on TPC-H and TPC-DS at SF 100. For TPC-H, we define one run as the consecutive execution of all 22 queries. As we have observed little slowdown for all TPC-DS queries as a whole in Section 3, we select the 20 queries with the highest measured slowdown to obtain a challenging workload for model-based placement. The 20 queries are executed consecutively for a single run. While running all queries sequentially may not be a realistic benchmark as the workload may be skewed towards few specific queries in reality, it is a good benchmark for model–based placement, as it results in a more diverse workload and a larger working set, making it harder to optimize the placement.

Placement mode and granularity. First, we compare the two placement modes, as well as heuristic-based placement from our earlier work [23]. For target–based placement, we set *target* to values between 0 % and 100 % in steps of 5 %. At *target* = 0 %, all data is placed in PMEM, and at *target* = 100 % all possible data, i.e., all data vectors, dictionaries, and inverted indices are placed in PMEM. Note that this does *not* result in 100 % of all column store data being placed in PMEM: SAP HANA automatically creates *block indices* for certain compressed columns [26], which could not be made page-loadable at the time of writing, and thus cannot be placed in PMEM in our prototype. Block indices make up 13 % of the column store's memory footprint for TPC-H, and 2 % for TPC-DS. Because they are part of the column store and used in query processing, we include their memory footprint in the reported totals. For budget-based placement, we set cost budgets of 0, 0.5, 1, 2, 3, 4, 5, 10, 20, 30, 40 and 50 % additional CPU time. We also show target–based placement at column and table granularity. The results at column granularity may serve as an indication of model–based data placement's usefulness for column stores without domain encoding.

Figure 9 shows the predicted percentage of the total data volume placed in PMEM and the predicted CPU time relative to DRAM-only placement for TPC-H. At structure granularity, both modes enable a controlled trade-off between the space and runtime costs of the placement configuration. While the heuristic comes reasonably close to the pareto frontier lined out by the model–based placement modes, the clear advantage of model–based placement is the ability to choose *any* point on the pareto frontier, instead of being locked into a single configuration without certainty about its optimality. For the given workload it is also always possible to reasonably satisfy the requested memory target in target mode. This is due to the structure-grained nature of the placement, and the figure shows that this is not always possible if data is placed at column- or table-granularity. Especially for table-granular placement a clear stair-like pattern emerges, a result of the access costs and memory footprint being dominated by the `LINEITEM` and `ORDERS` tables. Placing one of those tables completely in PMEM results in sharp increases of the slowdown and PMEM–resident data volume.

Prediction quality. Next, we evaluate the prediction accuracy for the generated placement configurations. We use the memory target mode for this, because it maps out the pareto frontier more evenly and the placement decisions in both modes are equally optimal. *target* is set to values between 0 % and 100 % in steps of 5 %.

Figure 10 shows model predictions and measured values for TPC-H and TPC-DS. For both workloads, the model predicts the real-world slowdown well. The median absolute errors across all placement configurations are only 1.4 % and 0.4 % for TPC-H and TPC-DS, respectively. The maximum absolute errors are 8.1 % and 2.7 %, respectively. The benefit of model–based data placement in heterogeneous main memory is also clearly visible. In both cases it would be possible to place 75 % or more of the column store's total data volume in PMEM with a performance drop of less than 10 % compared to placing all data in DRAM. In contrast to placement with a 90 % target, placing all data in PMEM would result in performance degradations of 44 % for TPC-H and 30 % for TPC-DS. Note that the totals in Figures 9 and 10 do not include columns that are not

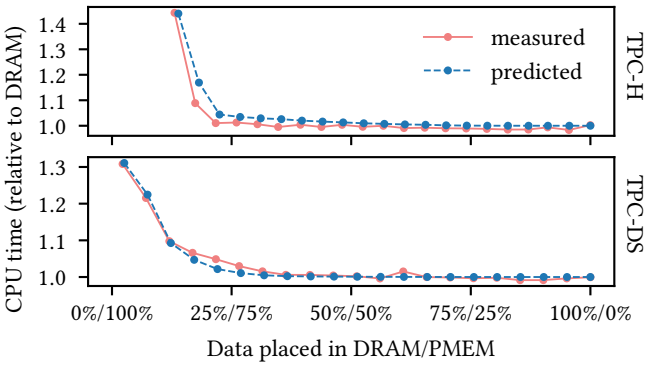

Figure 10: Predicted and measured performance for target–based placement of TPC-H and TPC-DS (SF 100).

accessed in the benchmark queries, e.g., `L_COMMENT`, `PS_COMMENT`, etc. for TPC-H, as those are never loaded into memory.

We also conducted the same experiment with far-NUMA PMEM. For both workloads the maximum slowdown increased by 6 % over the near-NUMA experiment. The median absolute model prediction errors were only 0.4 % for TPC-H and 0.9 % for TPC-DS. The maximum errors were 10.6 % and 3.9 %, respectively, showing that the model can adapt well to different memory latency characteristics.

It is also interesting to consider the resulting placement decisions: At the 90 % memory target for TPC-H, mostly dictionaries are kept in DRAM, especially those of `LINEITEM` and `ORDERS`. Inverted indices are all placed in PMEM, and in comparison to dictionaries only few data vectors are kept in DRAM. For TPC-DS the results are similar: Here, the dictionaries that are kept in DRAM mostly belong to the fact tables, while the data vectors remaining in DRAM are exclusively from dimension tables. This is likely because values often need to be materialized from these dimension tables requiring data vector point lookups which have much higher costs in PMEM. Note that this does not mean that simply placing *all* dictionaries in DRAM and the remaining structures in PMEM would be a good heuristic. For both workloads this policy would produce sub–optimal results, as it would leave dictionaries in DRAM that can be placed in PMEM with little performance impact, while placing data vectors in PMEM that do have performance impact. This underlines the benefits of using a cost model to optimize placement decisions over trying to find a blanket policy.

Comparison to Memory mode. Lastly, we compare App Direct mode to Memory mode using TPC-H. If PMEM's persistence is not needed, Memory mode could be a simpler option for cheaper and larger memory than using App Direct mode and managing data placement in software. For this experiment we replace the 512 GiB PMEM DIMMs in the **200 Series** system with 128 GiB PMEM DIMMs to obtain a configuration that is within the recommendation of a ratio of DRAM to PMEM between 1:4 and 1:16 for Memory mode [21]. As Memory mode uses the entire DRAM capacity of a socket as a hardware-managed L4 cache and the TPC-H SF 100 workload comfortably fits into this capacity on our test system, we use TPC-H SF 1000 as a second workload that exceeds DRAM capacity. For SF 1000, we omit Q09 and Q18 as the DRAM capacity was insufficient to process those queries. We compare CPU times for a consecutive run of all benchmark queries in Memory mode to two configurations in App Direct mode: PMEM-only placement, where all table data is placed in PMEM, and *optimal* placement, where we use model–based placement to place as much data as possible in DRAM while still leaving enough DRAM available for query processing. This represents the best attainable performance in App Direct mode using our approach. For SF 100, optimal placement can place all table data in DRAM, while for SF 1000 only 50 GiB can be placed in DRAM, roughly 20 % of all referenced table data.

Relative to the optimal placement, PMEM-only placement in App Direct is roughly 50 % slower for SF 100, and 130 % slower for SF 1000. Running the workload in Memory mode has the same performance as optimal placement for SF 100. This is unsurprising as optimal placement for SF 100 means that all data is placed in DRAM, and the workload also fits into the DRAM cache in Memory mode. For SF 1000, we expected Memory mode to perform worse than App Direct mode as it would have to fall back to PMEM for many memory requests with the workload's memory footprint exceeding the DRAM cache capacity [18]. Including table data and intermediate results, the workload's peak memory utilization was roughly 300 GiB, exceeding the 128 GiB DRAM capacity. However, running the SF 1000 workload in Memory mode is 18 % faster than optimal placement in App Direct mode. The reason for this is the following: As no query in the benchmark ever references all table data, the working set at any given time still fits into the DRAM cache, and Memory mode can thus perform very close to real DRAM.

This suggests that it may be advantageous to use Memory mode for workloads that have a large overall footprint, but a working set that can fit into the DRAM cache most of the time. However, Memory mode also has multiple disadvantages over App Direct mode: First, the total memory capacity is reduced, as DRAM becomes a hardware-managed cache. Second, PMEM's persistence feature cannot be utilized. Memory mode also makes it impossible to pin data in DRAM, which precludes prioritizing important jobs as all memory accesses in the system put pressure on the DRAM cache. Lastly, it is only supported for 2-socket systems [20, 27]. So on 4-socket systems or if one wants to utilize the persistence feature, e.g., for low-latency logging and faster restart times [2, 17, 37], Memory mode may not be an option. Effectively using App Direct mode in such scenarios further beyond what we have shown here is an interesting topic for future work. One possible direction could be to combine our cost model with buffer–managed approaches that move data between DRAM and PMEM [46, 49] to dynamically adapt the DRAM-resident data to the workload.

Discussion. While we have focused on evaluating the approach of using logical access counters and microbenchmarks of access primitives to obtain a cost model for data placed in PMEM in SAP HANA, we believe that it would be equally applicable to other IMDBMS such as MonetDB, HyPer, or Hyrise. It may not be possible to obtain equally optimal results in systems that do not employ domain encoding (cf. Figure 9), but this does not inherently limit general applicability. For systems that horizontally partition data such as Hyrise, it may also be beneficial to count accesses per column partition and to place data at that granularity. If a system is

more likely to run into bandwidth limitations, like we have observed for MonetDB (cf. Section 3.3), it may also be necessary to extend the model with a bandwidth term to keep its predictions suitably accurate for making placement decisions. We believe that our modelling approach may also be useful in utilizing upcoming heterogeneous memory technologies such as disaggregated memory using CXL. In this context it may also be worthwhile to extend the approach to consider data placement on more than two types of memory, e.g., local DRAM, local PMEM, CXL-attached DRAM, and CXL-attached PMEM, to optimize the cost-performance tradeoff.

Lastly, in this work we have focused exclusively on the placement of *table* data in PMEM. Future work should also consider using PMEM as cheaper and larger memory for *temporary* data. As our comparison to Memory mode showed, it is difficult to use PMEM effectively for large OLAP workloads, as the smaller DRAM capacity easily becomes the bottleneck for workload size if PMEM cannot be utilized for intermediate results.

6 RELATED WORK

Cost-efficient data systems. Using PMEM as a cheaper alternative to DRAM is related to a larger body of recent work on cost-efficient data systems [24, 25, 31, 36]. For example, LeanStore [24] and Umbra [36] show that disk-based systems can approach in-memory performance at much lower costs. Making effective use of higher-capacity, lower-cost media than DRAM while keeping in-memory performance is highly desirable in this context. In this work, we take a commercial in-memory DBMS as the starting point and show how a cost model can be used to make optimal data placement decisions that reduce costs while largely keeping performance on par with the original in-memory setting. Our approach is inspired by Vogel et al.'s extension to Umbra, Mosaic [47], where a cost model is used for pareto–optimal data placement in a pool of *storage devices* for scan-heavy relational workloads.

PMEM in data management. Many works focus on designing or optimizing data structures for PMEM [3, 9, 29, 32, 38] or redesigning DBMS components and architecture with PMEM's persistence features in mind [4, 28, 37]. We concentrate on efficiently placing existing, read-optimized storage data structures of a commercial DBMS in PMEM, motivated by its lower cost and higher capacity compared to DRAM. Other studies also evaluate PMEM as larger and cheaper volatile memory for IMDBMS data: Early work by Shanbag et al. [42] reports a 60 % mean slowdown of the Star Schema Benchmark when data is placed in PMEM instead of DRAM. Avni et al. [5] benchmark in-memory OLTP on PMEM and observe little performance impact for a read-heavy variant of TPC-C, similar to our results. For full TPC-C including updates that perform writes to PMEM, they observe reduced throughput by up to 3.5× depending on which data is placed in PMEM. For integrating PMEM in the storage layer, buffer-managed approaches have been proposed [12, 46, 49], where PMEM is used as an additional tier in the cache hierarchy. These approaches base their eviction and migration policies on when a page was last accessed, and may duplicate data across DRAM and PMEM. We instead view storing data in PMEM as a placement problem where data resides either in PMEM or in DRAM and propose a cost model to guide the placement decision. However, combining a buffer-managed approach with our cost model is an interesting topic for future work. Work on placing data across other levels of the memory hierarchy based on counting random and sequential accesses also exists: Bhattacharjee et al. [6] consider static data placement in an HDD-SSD storage hierarchy and find that the SSD should first be used for data that is mostly randomly accessed. They also express the placement problem as a knapsack problem, but use a greedy heuristic to find its solution. Boissier et al. [7] make placement decisions between DRAM and secondary storage at column-granularity, using a different layout for data placed in secondary storage. We also distinguish random point and sequential scan accesses in this paper, but for placing data in DRAM or PMEM, and without changing the storage layout.

Memory cost modelling. Modelling the cost of memory accesses for data management tasks is also a topic of prior work: Initial work by Manegold et al. [33, 34] proposes modelling memory access costs for database workloads by approximating cache misses and scoring them by their latency, similar to our approach. However, approximating cache misses requires extensive modelling of the access patterns used by query processing algorithms. In contrast, we rely on counting logical accesses to characterize workloads and use microbenchmarks to obtain a black-box model for scoring these accesses, simplifying the resulting cost model. Clapp et al. [10, 11] show how to predict changes in performance for changing memory latencies and bandwidths for compute- and memory-intensive workloads in general. To support this wide range of workloads, they use hardware counters for workload characterization. In our case, applying domain knowledge about the column store allows us to use logical access counters in software instead, with the advantage of being able to attribute costs to individual structures and to guide placement of the structures in a hybrid memory system.

7 CONCLUSION

We have evaluated the performance impact of placing in-memory column store table data in PMEM instead of DRAM, motivated by PMEM's lower cost and higher capacities. For analytical workloads like TPC-H and TPC-DS we observed queries taking up to 2.2× more CPU time if all data is placed in PMEM due to its higher read latencies compared to DRAM, while the impact on OLTP-like point queries is less noticeable. We argued that for analytical workloads a much better cost-performance tradeoff can be achieved by carefully selecting data structures to be placed in DRAM or PMEM. To this end, we proposed a cost model to predict the workload slowdown for arbitrary placement configurations and discussed how this model can be used for targeted data placement. Our evaluation shows that the model accurately predicts slowdowns across the generated placement configurations and allows placing more than 75 % of table data in PMEM while keeping the slowdown relative to the DRAM-only configuration below 10 % for the analytical workloads. If one does not require PMEM's persistence, an initial comparison to Memory mode, which exposes PMEM only as volatile memory and uses DRAM as a hardware-managed cache, has shown that this mode may provide better performance than model-driven placement in App Direct mode.

ACKNOWLEDGMENTS

We would like to thank Intel for providing the test systems.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[2] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA adoption of non-volatile memory. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1754–1765. https://doi.org/10.14778/3137765.3137780

[3] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3164135.3164147

[4] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1753–1758. https://doi.org/10.1145/3035918.3054780

[5] Hillel Avni, Aharon Avitzur, Nir Pachter, and Vladi Vexler. 2021. Extending in-memory OLTP with persistent memory. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*. VLDB Endowment, 10–16. http://www.adms-conf.org/2021-camera-ready/avni_adms21.pdf

[6] Bishwaranjan Bhattacharjee, Mustafa Canim, Christian A Lang, George A Mihaila, and Kenneth A Ross. 2010. Storage class memory aware data management. *IEEE Data Eng. Bull.* 33, 4 (2010), 35–40.

[7] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Piscataway Township, NJ, USA, 209–220. https://doi.org/10.1109/ICDE.2018.00028

[8] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. 2021. Precise, compact, and fast data access counters for automated physical database design. In *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Gesellschaft für Informatik, Bonn, 79–100. https://doi.org/10.18420/btw2021-04

[9] Shimin Chen and Qin Jin. 2015. Persistent B^+-Trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[10] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, Piscataway Township, NJ, USA, 213–224.

[11] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. A simple model to quantify the impact of memory latency and bandwidth on performance. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, New York, NY, USA, 471–472.

[12] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya Dulloor. 2014. A prolegomenon on OLTP database systems for non-volatile memory. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014*. VLDB Endowment, 57–63.

[13] Ulrich Drepper. 2007. *What every programmer should know about memory*. Red Hat, Inc. Retrieved 2022-05-31 from https://akkadia.org/drepper/cpumemory.pdf

[14] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1206–1220.

[15] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database–an architecture overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[16] Google. 2016. *Benchmark: A microbenchmark support library*. Google. Retrieved 2022-05-31 from https://github.com/google/benchmark

[17] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 877–892.

[18] Mark Hildebrand, Julian T Angeles, Jason Lowe-Power, and Venkatesh Akella. 2021. A Case Against Hardware Managed DRAM Caches for NVRAM Based Systems. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 194–204.

[19] Intel. 2019. *Intel® Optane™ DC Persistent Memory product brief*. Intel. Retrieved 2022-05-31 from https://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf

[20] Intel. 2021. *Intel® Optane™ Persistent Memory 200 Series product brief*. Intel. Retrieved 2022-05-31 from https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf

[21] Intel. 2021. *Intel® server board M50CYP2SB family technical product specification*. Intel. Retrieved 2022-05-31 from https://www.intel.com/content/dam/support/us/en/documents/server-products/single-node-servers/m50cyp2sb-server-board-tps.pdf

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019), 61. arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[23] Robert Lasch, Robert Schulze, Thomas Legler, and Kai-Uwe Sattler. 2021. Workload-driven placement of column-store data structures on DRAM and NVM. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. Association for Computing Machinery, New York, NY, USA, 1–8.

[24] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[25] Viktor Leis and Maximilian Kuschewski. 2021. Towards cost-optimal query processing in the cloud. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1606–1612.

[26] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. 2010. Speeding up queries in column stores. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, Springer, Berlin, Heidelberg, 117–129.

[27] Lenovo. 2022. *Intel Optane Persistent Memory 200 Series Product Guide*. Lenovo. Retrieved 2022-05-31 from https://lenovopress.lenovo.com/lp1380-intel-optane-persistent-memory-200-series

[28] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. 2019. Persistent buffer management with optimistic consistency. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 3 pages. https://doi.org/10.1145/3329785.3329931

[29] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[30] Jason Liu, Pedro Espina, and Xian-He Sun. 2021. A study on modeling and optimization of memory systems. *Journal of Computer Science and Technology* 36, 1 (2021), 71–89.

[31] David Lomet. 2018. Cost/performance in modern data stores: How data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, New York, NY, USA, 1–10.

[32] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[33] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730.

[34] Stefan Manegold, Peter Boncz, and Martin L Kersten. 2002. Generic database cost models for hierarchical memory systems. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, Elsevier, Amsterdam, The Netherlands, 191–202.

[35] MemVerge. 2020. *More Memory. Less Cost*. MemVerge. Retrieved 2022-05-31 from https://memverge.com/more-memory-less-cost/

[36] Thomas Neumann and Michael J Freitag. 2020. Umbra: A disk-based system with in-memory performance. In *CIDR*. www.cidrdb.org, Amsterdam, The Netherlands, 7. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[37] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (Snowbird, Utah) *(DaMoN '14)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2619228.2619236

[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[39] Hasso Plattner and Alexander Zeier. 2012. *In-memory data management: technology and applications*. Springer Science & Business Media.

[40] PLDA. 2021. *Reducing CXL Latency with PLDA and AnalogX*. Retrieved 2022-05-31 from https://www.plda.com/blog/category/technical-article/reducing-cxl-latency-plda-and-analogx

[41] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the latency gap between NVM and DRAM for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, New York, NY, USA, 1–8.

[42] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-scale in-memory analytics on Intel® Optane™ DC persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, New York, NY, USA, 1–8.

[43] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, et al. 2019. Native store extension for SAP HANA. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2047–2058.

[44] Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K Goel, Anisoara Nica, Peter Bumbulis, Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, et al. 2016. Page as you go: Piecewise columnar access in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1295–1306.

[45] Túlio A. M. Toffolo and Haroldo G. Santos. 2022. *Python-MIP*. Python-MIP. Retrieved 2022-05-31 from https://www.python-mip.com

[46] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1541–1555. https://doi.org/10.1145/3183713.3196897

[47] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. 2020. Mosaic: A budget-conscious storage engine for relational database systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2662–2675.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Berkeley, CA, USA, 169–182.

[49] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*. 2195–2207.