Analyzing Memory Accesses With Modern Processors

Stefan Noll
stefan.noll@sap.com
TU Dortmund University
SAP SE

Jens Teubner
jens.teubner@cs.tu-dortmund.de
TU Dortmund University

Norman May
norman.may@sap.com
SAP SE

Alexander Böhm
alexander.boehm@sap.com
SAP SE

ABSTRACT

Debugging and tuning database systems is very challenging. Using common profiling tools is often not sufficient because they identify the machine instruction rather than the instance of a data structure that causes a performance problem. This leaves a problem's root cause such as memory hotspots or poor data layouts hidden. The state-of-the-art solution is to augment classical profiling with a memory trace. However, current approaches for collecting memory traces are not usable in practice due to their large runtime overhead.

In this work, we leverage a mechanism available in modern processors to collect memory traces via hardware-based sampling. We evaluate our approach using a commercial and an open-source database system running the JCC-H benchmark. In particular, we demonstrate that our approach is practical due to its low runtime overhead and we illustrate how memory traces uncover new insights into the memory access characteristics of database systems.

ACM Reference Format:
Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2020. Analyzing Memory Accesses With Modern Processors. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3399666.3399896

1 INTRODUCTION

Today's database management systems are increasingly complex and complicated [40]. They offer numerous features, configuration parameters, and low-level optimizations for various hardware setups. This complexity makes it difficult to debug a system during development or to identify new optimization and tuning opportunities.

To gain insights into the execution engine and data flow, engineers rely on general-purpose profiling tools such as perf [20] or VTune Profiler [14], or on custom profiling mechanisms implemented directly into the DBMS [31]. Common profiling tools pinpoint the machine *instruction* (and the source code line) where CPU time is spent or where hardware events such as cache misses occur. However, profiling tools often fail to identify the *instance of a data structure* that causes a problem, which makes a root cause analysis very challenging. In fact, whenever the same program code such as `hash_table.lookup(key)` is executed for different instances of a hash table, it may be impossible to detect the instance (e.g., hash table used in hash join of R and S) that causes problems—even with detailed call stack information. In addition, performance problems might only occur when accessing some parts of a data structure, e.g., due to skew.

To identify the root cause of a performance problem, others [15, 16, 23, 27, 35] propose to combine profiling information with a *memory trace*, i.e., all memory addresses the system accesses during runtime. By assigning profiling metrics such as CPU time or cache misses to memory addresses rather than machine instructions, we can identify specific instances or parts of a data structure that cause performance problems.

However, collecting memory traces with tools such as Valgrind [24] or Intel's Pin [21] incurs a high overhead: They slow down the application by more than an order of magnitude. This makes them unusable for profiling complex applications such as database systems—especially for analyzing issues that only occur in production. The good news is that modern processors feature powerful profiling capabilities via *precise event-based sampling* (PEBS) [13] that potentially allows overcoming these restrictions.

In this work, we demonstrate that collecting memory traces with PEBS on recent Intel processors is feasible in practice. We show in a comprehensive experimental evaluation that memory traces provide detailed information about

DAMON'20, June 15, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8024-9/20/06…$15.00
https://doi.org/10.1145/3399666.3399896

how a database system accesses memory. We analyze the access frequency and the access pattern of memory accesses to reveal skew, hot data structures, or implementation and algorithmic details of the execution engine. In particular,

(i) we present a practical implementation of memory tracing based on Intel's PEBS mechanism;
(ii) we evaluate our approach using both a commercial and an open-source database system running the JCC-H benchmark;
(iii) we demonstrate and discuss practical use cases; and
(iv) we analyze the runtime overhead.

We present our first major contribution, the memory tracing implementation, in Section 3. In Section 4, we evaluate the implementation and discuss practical use cases as our second and third major contribution. As our final major contribution, we study the runtime overhead in Section 5.

2 BACKGROUND

Profiling. Common profiling tools such as perf [20] or VTune Profiler [14] allow analyzing detailed performance characteristics of applications. They incur almost no slowdown. This makes them usable everywhere—even in production environments. In addition to detecting where CPU time is spent, they also provide microarchitectural insights by collecting events that are exposed by modern processors via hardware performance counters [13]. These allow the user to measure cache misses, stalled cycles, memory bandwidth, non-uniform memory accesses, TLB misses, and many more events [13]. Profilers can report the machine instruction (and source code line) that was executed when an event occurred. They do *not* reveal what and how *data* was accessed.

VTune Profiler takes a step into this direction. It features a profiling mode that maps certain events such as cache misses to memory objects by instrumenting memory allocations and de-allocations [15]. However, VTune Profiler does not provide a memory trace, which is necessary to reveal detailed memory access statistics and access patterns.

Precise Event-Based Sampling. We use hardware performance counters with support for *precise event-based sampling* (PEBS) [13]. PEBS is available in modern Intel processors and currently supports a subset of the events such as L1, L2, L3 cache misses or cache hits, all memory reads, or all memory stores. PEBS enables writing debug information associated with an event to a memory resident buffer. In addition to a precise instruction pointer, this information includes, e.g., copies of general-purpose registers, latency information or the accessed data address. Figure 1 illustrates the mechanism.

The operating system configures the *performance monitoring unit* (PMU) of a processor's core to count an event such as an L2 cache miss. It specifies the sampling rate by setting a threshold and creates a buffer for PEBS records in memory. Then, the PMU counts the specified event and, when the counter reaches the threshold, the hardware automatically writes a record with debug information to the buffer. When the buffer is full, the PMU sends an interrupt. The interrupt triggers the interrupt handler of the operating system. The interrupt handler drains the buffer and processes the records. Afterwards, it resets the counter and the PMU starts counting again. The advantage is that the *hardware* writes the information to memory. In addition, the buffer mechanism amortizes the cost of executing the interrupt handler.

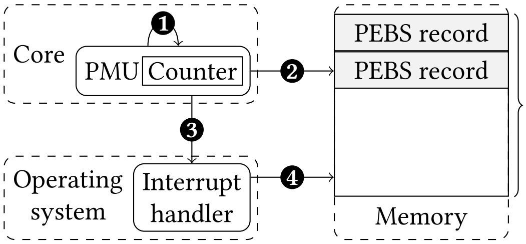


Figure 1: PEBS mechanism. The core's PMU counts an event such as L2 cache misses ❶. It writes a record to memory when the $\boxed{\text{Counter}}$ reaches a threshold ❷. It sends an interrupt when the buffer is full ❸. Then, the interrupt handler of the operating system drains the buffer ❹ and processes the records.

3 MEMORY TRACING

The memory tracing implementation is our first major contribution of this work. To minimize runtime overhead, we leverage the PEBS mechanism of modern processors (cf. Section 2) to trace memory accesses. The PEBS mechanism allows us to sample the memory address associated with a specific hardware event. For the experiments presented in this work, we focus on the event `mem_load_uops_retired.all_loads`, which occurs whenever the CPU *reads* data from memory [13]. This includes both cache hits and cache misses to the L1, L2 and last-level cache. Note that depending on the use case, we could use other events to collect memory addresses, e.g., associated with all memory *writes*, only last-level *cache misses*, or cache misses where the cache line was modified by another core (possibly indicating *false sharing* [23]).

Figure 2 gives an overview of our approach. We assume that the PEBS mechanism of the PMU is already configured to write a record every *n*-th occurrence of the event. For each logical core, we create a buffer for storing the address samples. When a worker thread of the DBMS accesses memory, the core's PMU may write a record with the event's debug information to the record buffer (cf. Figure 1). When the record buffer is full, the PMU triggers the operating system which executes a custom interrupt handler to process the collected records. It extracts only the field with the virtual

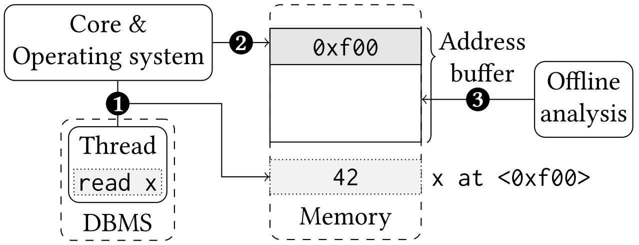


Figure 2: Our memory tracing implementation. When a thread accesses a data structure ❶, we may collect the virtual memory address of the accessed data using PEBS (cf. Figure 1) and store the address in memory ❷. We analyze the addresses offline ❸.

memory address associated with the event and stores the address into the buffer of the logical core. After running a workload, we analyze the address data.

To enable memory tracing in performance-critical environments, we implement the memory tracing by modifying the Linux kernel (version 5.1) and by adding a custom kernel module[1]. Our implementation has ~1000 lines of code. In particular, we leverage the extensive, tested functionality of the perf subsystem [41] of the Linux kernel to program a core's PMU, to setup PEBS, to register an interrupt handler, and to implement filtering, e.g., for user space or kernel space, or for particular processes or threads. We can start and configure the memory tracing using the perf tool from user space. To improve scalability, we modify the interrupt handler to place the sampled addresses into per-core address buffers instead of the global ring buffer used by the perf subsystem. The kernel module acts as an interface for managing the address buffers from user space.

Modifying the perf subsystem is necessary in order to reduce the runtime overhead of collecting memory addresses with a high sample frequency. That is because the perf subsystem collects extensive metadata for a single sample. Processing this metadata wastes memory and compute time. In contrast, our implementation is very efficient: It extracts and stores only the addresses needed for memory tracing.

4 USE CASES

The evaluation of our implementation of memory tracing and the demonstration of practical uses cases for database systems are the second and third major contribution of this work. Note that we expect many more use cases, e.g., when focusing on other parts of the system such as intermediate results or when using other hardware events.

Experimental Setup. We use the JCC-H benchmark [4], an extension of the TPC-H benchmark [39] with skewed data and query predicates, with a scale factor of 10. We execute custom queries as well as a complete workload with 200 random queries of the JCC-H benchmark (excluding Q21). We run SAP HANA [10], a commercial, in-memory database management system. SAP HANA makes heavy use of order-preserving dictionary compression[2]—similar to other systems [5, 17, 18, 28]. In addition, we run experiments with DuckDB [29], an open-source, embedded, analytical database system, by using its Python interface. Our test machine has two Intel Xeon E5-2670 v3 processors and 256 GB of main memory. While DuckDB executes queries with a single thread, SAP HANA executes queries with multiple threads using up to 48 logical cores.

Processing of Memory Traces. Due to the sampling mechanic we do not trace every memory load, especially if data is accessed only once during profiling. To compensate for missed accesses due to sampling and to improve visualization, we *group addresses into buckets* of a fixed size. We denote the bucket size in each figure. This means that, for each memory address, we report a data access of, e.g., 4 KiB instead of 8 bytes[3]. We assign an address to a bucket of size, e.g., 4 KiB by ignoring the least significant $\log_2(4096) - 1 = 11$ bits of the address. We explain the visualization of a memory trace using the example of Figure 3 in Section 4.1.

4.1 Detecting Access Patterns

To illustrate how memory traces reveal access patterns or other implementation and algorithmic details of the execution engine, we analyze the execution of a custom query with DuckDB. The SQL statement is shown below:

```
SELECT o_totalprice, o_orderdate, o_shippriority
FROM orders WHERE o_orderstatus = 'O'
ORDER BY o_totalprice
```

Figure 3 shows the memory trace. It visualizes how DuckDB accesses memory over time. The x-axis shows the samples ordered by their sampling time. The y-axis represents the virtual memory address of the samples sorted by address in ascending order. We illustrate the size of the accessed memory. In particular, we visualize each sampled address as an access to a bucket of size 4 KiB.

Figure 3 shows that scanning the table ⓐ while applying the filter predicate ① reads memory *sequentially*. Afterwards, DuckDB sorts ② the data. The trace reveals that the sort operator accesses one data structure *sequentially*

[1] The source code is available at `http://dbis.cs.tu-dortmund.de` and at `https://github.com/stefannoll/mat`.

[2] An ordered dictionary maps domain values to a dense set of consecutive numbers. Instead of the actual value of the columns, the engine stores the typically much smaller index of the dictionary entry. The encoded column or the dictionary may be further compressed.

[3] The actual data size depends on the load instruction associated with the sampled memory address: e.g., 8 B for 64-bit or 4 B for 32-bit operations.

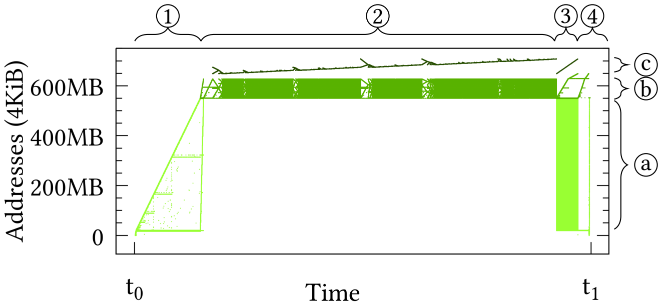
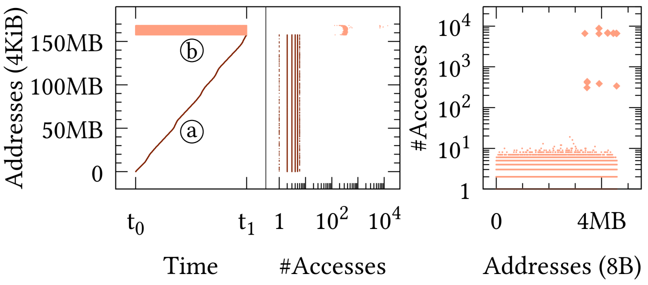


Figure 3: The trace illustrates the access patterns of DuckDB's execution engine. In particular, it reveals how the operators for filtering ①, sorting ②, materializing ③, and assembling the result ④ access important data structures: the table (■) ⓐ, the filtered column (■) ⓑ, and the position list (■) ⓒ.



(a) Column and dictionary. **(b) Dictionary only.**

Figure 4: The aggregation operator of SAP HANA accesses the encoded column (■) ⓐ *sequentially*. Due to the data distribution, it accesses the dictionary (■) ⓑ *randomly* (a). The trace reveals skew at the granularity of individual dictionary entries (◆) (b).

and the other *randomly*. Note that DuckDB's implementation uses the quicksort algorithm and that it sorts a position list ⓒ instead of the actual data. To compare a position `p`, it accesses the filtered column ⓑ indirectly by fetching the value with `column[p]`, which may be increasingly random as the list's order changes. The trace illustrates both access patterns: The quicksort algorithm splits the position list recursively and traverses each sublist from the start and the end simultaneously; for the comparison, it indirectly accesses the column. In the next phase, DuckDB materializes ③ the projected columns. It reads the sorted position list *sequentially* and accesses the table *randomly*. Finally, the Python interface of DuckDB transforms the result ④ into Python data structures.

Note that the memory trace allows us to break down the individual phases/operators of the query execution. We can infer the different memory access patterns from the visualization alone. We do *not* require the source code to collect the trace. Knowing the memory addresses of data structures by tracking allocations or knowing the implemented algorithms helps, however, to explain the trace.

4.2 Access Counting at Byte-level

To demonstrate how memory traces allow us to collect detailed access statistics at byte-level and to reveal skew, we analyze the execution of a custom query with SAP HANA:

```
SELECT AVG(l_extendedprice) FROM lineitem
```

We limit SAP HANA to execute the query with two threads. We show only the results of the first thread (the results of the other thread are very similar). Figure 4 illustrates the memory trace of the encoded column and the dictionary of `l_extendedprice`. Figure 4a shows the memory accesses over time (left) and the total number of accesses as a histogram (right). Figure 4b visualizes the total number of accesses to only the dictionary at byte-level. The samples are sorted by address in ascending order.

The aggregation operator *sequentially* reads the encoded column. Due to the dictionary encoding, it needs to decode each reference in the column by looking up its value in the dictionary. The data distribution causes these accesses to be *random*. In addition, we observe that the dictionary is accessed more frequently (`l_extendedprice` contains 97.77 % duplicates). This demonstrates that memory traces show both the access pattern and the access frequency in detail.

When we look only at the dictionary, the trace reveals that the dictionary accesses are heavily skewed: 20 entries are accessed more frequently than others (by several orders of magnitude). Note that this property of the JCC-H benchmark becomes easily observable with the memory trace: We are able to identify "hot" data at the granularity of *memory loads*—not only at the granularity of pages [11]. By tracking memory allocations in SAP HANA, we know the memory address range of the dictionary. This allows us to identify, for example, that the dictionary entry at position 997959 (with the value 55740.45) has the most accesses.

4.3 Hot Working Set Size

To demonstrate how memory traces enable us to estimate a workload's "hot" working set size, we execute a workload with 200 random queries of the JCC-H benchmark with SAP HANA. To evaluate a smaller data set, we also run a modified version of the workload with only 85 out of 200 queries that do not reference the `orders` table. SAP HANA executes the query workload on all 48 logical cores. Figure 6a visualizes the memory trace as a histogram, where we sort the sampled addresses by how often they occur in the trace.

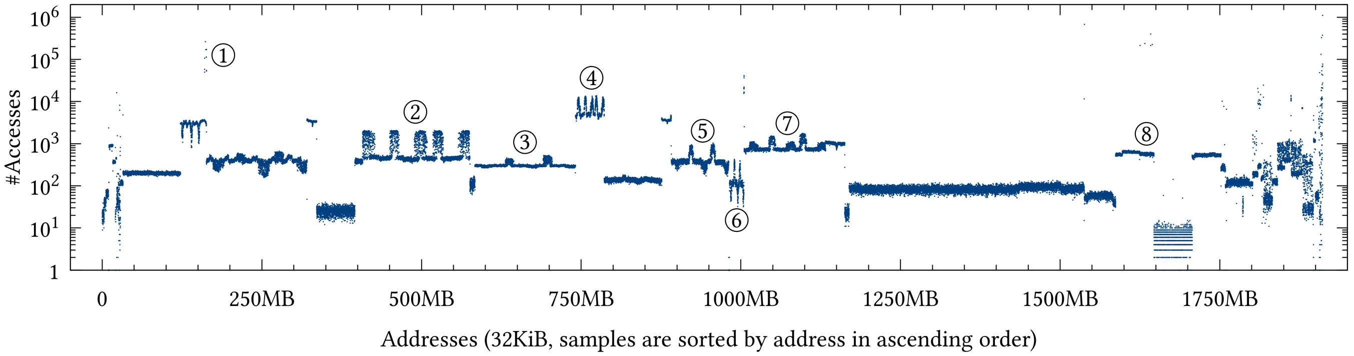
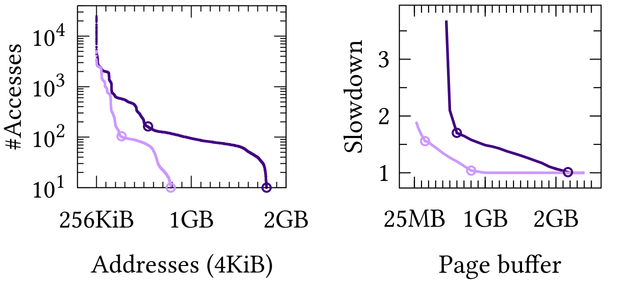


Figure 5: JCC-H benchmark with 200 random queries. The memory trace details the access pattern of the table data and reveals skew. We detect the 5 "populous orders", e.g., for the encoded columns `l_orderkey` ② and `l_quantity` ④. We also detect filter skew that causes 2 of the populous orders to be accessed more frequently, e.g., for the encoded columns `l_partkey` ③, `l_shipdate` ⑤, `l_shipmode` ⑥, and `l_suppkey` ⑦.



(a) Histogram sorted by number of accesses.

(b) Slowdown for different buffer sizes.

Figure 6: JCC-H benchmark with all tables (■) and without the `orders` table (■). The working set size associated with a specific access frequency (a) matches (○) the performance characteristics of executing the workload with a specific page buffer size (b).

We observe that the complete workload accesses table data (encoded columns and dictionaries) with a size of 1.8 GB. In contrast, the total size of the table data of all tables amounts to 3.8 GB in main memory. This demonstrates that the memory trace allows us to quantify the *working set size*, i.e., the size of the table data that is *actually* accessed during the execution of the benchmark. Additionally, we can measure how much data the workload accesses with a specific frequency, i.e., the "hot" data. We discover, for example, that the system accesses table data with a size of 600 MB more frequently.

Deriving a Buffer Size. We can use this information to derive a buffer size for SAP HANA when we execute the workload with page-loadable columns [33], i.e., using a page buffer to hold only a subset of the table data—at page granularity—in memory. Figure 6b shows how the size of the page buffer impacts execution time: It illustrates the relative slowdown compared to the execution time when all data fits in memory. If we compare the results to the working set size of a specific access frequency, shown in Figure 6a, we observe a strong similarity (highlighted in the figures with **○**). The same holds true for the workload without the `orders` table.

We argue that the traces could help to determine the optimal buffer size for disk-based systems [25] or help to size DRAM when using NVRAM as main memory and DRAM only as a cache—referred to by Intel as "memory mode" [12].

4.4 Table Partitioning

To demonstrate how memory traces allow us to analyze table partitioning, we use again the JCC-H workload consisting of 200 queries. SAP HANA executes the query workload on all 48 logical cores. Figure 5 shows the memory trace as a histogram, where the samples are sorted by address in ascending order. The memory trace illustrates the access pattern of the encoded columns and dictionaries.

The memory trace reveals the *populous order*[4] skew [4] of the table data. The skew becomes visible in the access pattern of the encoded column `l_orderkey` ② and `l_quantity` ④, where 5 parts of the columns (the 5 populous orders) are accessed more frequently. In addition, the trace also reveals filter skew [4]. In the access pattern of the encoded columns `l_partkey` ③, `l_shipdate` ⑤, `l_shipmode` ⑥, as well as `l_suppkey` ⑦, we observe that only 2 parts of the column (2 of the 5 populous orders) are accessed more frequently. The reason is that query predicates include the years 1993 and 1994 more frequently, resulting in more accesses to the 2 populous orders of the two years. The trace also highlights the skew of `l_extendedprice`, where 20 distinct values of the dictionary ① are accessed more frequently (cf. Section 4.2).

[4] The JCC-H benchmark has 5 populous orders with 25 % of the lineitems.

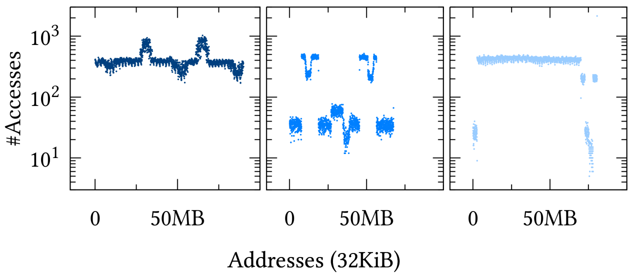


Figure 7: Impact of partitioning on `l_shipdate`. Comparison of no partitioning (■), partitioning per year (■) and partitioning per populous order (■).

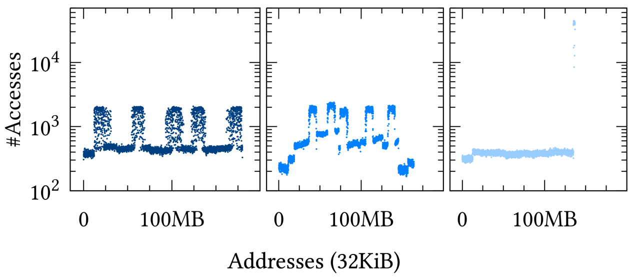


Figure 8: Impact of partitioning on `l_orderkey`. Comparison of no partitioning (■), partitioning per year (■) and partitioning per populous order (■).

Impact of Partitioning. We can use the trace to analyze the impact of table partitioning. We compare no partitioning (previously shown in Figure 5) to the partitioning used by Microsoft SQL Server 2017 for the TPC-H benchmark [7]. They recommend a range partition on the columns `o_orderdate` and `l_shipdate`. We perform a range partition per *year* on the two columns which splits the tables `lineitem` and `orders` in 7 partitions each. Furthermore, we compare to a range partition on the column `l_orderkey`, which splits the `lineitem` table in 6 partitions: a partition per *populous order* and one partition holding the remaining rows. Figures 7, 8, and 9 visualize how the different partitioning schemes change the access pattern of the encoded columns `l_shipdate` ⑤, `l_orderkey` ②, and `o_orderdate` ⑧.

We observe that the partitioning causes some parts of the columns to be accessed rarely, i.e., they become "colder". Instead of focusing only on execution time, the trace enables us to evaluate the partitioning schemes by *quantifying* the accessed data volume (by multiplying the number of accesses per bucket with the bucket size). The partitioning per *year* decreases the accessed data volume for `l_shipdate`, `l_orderkey`, and `o_orderdate` by 72 %, 5 %, and 93 %. The partitioning per *populous order* decreases the accessed data volume by 19 %, 15 %, and 0 %, respectively. While the partitioning per year allows for partition pruning whenever a filter predicate selects only some years, the partitioning per populous order increases the access locality of the `lineitem` table for populous orders. We observed similar effects for other columns (not shown).

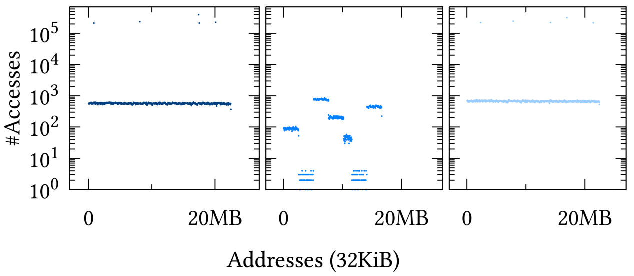


Figure 9: Impact of partitioning on `o_orderdate`. Comparison of no partitioning (■), partitioning per year (■) and partitioning per populous order (■).

5 RUNTIME OVERHEAD

The analysis of the runtime overhead of our implementation is the final contribution of this work. The advantage of using Intel's PEBS mechanism for memory tracing is that the *hardware* writes sampled addresses automatically to memory and that it buffers the samples. The user can configure the sampling rate to adjust the trade-off between runtime overhead and precision by setting the threshold that controls after how many events the CPU writes a PEBS record (cf. Section 2). Table 1 shows how the threshold impacts the execution time of the JCC-H workload of 200 queries running on SAP HANA on all 48 logical cores.

Threshold	200	400	600	800	1000	2000	4000
Slowdown	2.30×	1.67×	1.45×	1.34×	1.27×	1.13×	1.05×
Trace [GB]	45.192	22.999	15.393	11.585	9.285	4.656	2.324
Trace* [GB]	1.518	0.855	0.548	0.375	0.308	0.134	0.064

Table 1: Tracing overhead for different thresholds: the slowdown of the execution time, the size of the complete trace, and the size of the trace containing only addresses of table data, i.e., of the encoded columns and the dictionaries (*).

In our experiments presented in Section 4.4, we use a threshold of 1000 which causes a slowdown of 27 %. In the other experiments, we use a threshold of 200 to demonstrate a very high precision. It increases runtime by a factor of 2.3. Our implementation based on PEBS is more than an order of magnitude faster than approaches based on binary instrumentation such as Valgrind [24] or Intel's Pin [21]. It makes memory tracing practical and even usable in production. The user can further decrease overhead by lowering the sampling rate or by choosing a hardware event which occurs less frequent, e.g., only cache misses.

6 RELATED WORK

Tracing Methods. Related work from the systems community explores different approaches for collecting memory traces: simulation or emulation [3], binary instrumentation [6, 9, 21, 24], passing all memory access through an FPGA [19], or using custom hardware to snoop the memory bus of DRAM DIMMs [2]. These approaches allow tracing *all* memory accesses at the cost of slowdowns by more than an order of magnitude or require additional hardware. Others propose to use performance monitoring units of modern processors from AMD [16], Intel [32], or IBM Power [36] to trace memory accesses via hardware-based *sampling*. The integration effort or the runtime overhead of employing such an approach in practice remains unclear, however. Other related work [1, 26] studies PEBS parameters such as the size of the record buffer and the sampling rate—but not for an end-to-end database workload. To the best of our knowledge, we are the first to explore memory tracing via PEBS running (a complex workload on) a commercial database system.

Use Cases. While we use memory tracing to study access patterns, to detect skew or to count accesses, related work explores many other use cases. Tozun et al. use memory traces together with hardware simulation to map cache misses back to database operators and to components of the storage manager for OLTP workloads [38]. They propose a transaction scheduling mechanism that uses memory traces to maximize instruction cache locality [37]. Others use memory traces to build cache miss ratio curves to quantify an application's cache usage [36] or to derive cache partitioning schemes [42], to detect memory errors such as buffer overflows or use-after-frees [34], to detect false sharing [30], to optimize data placement in NUMA systems [8, 22], to pinpoint performance bottlenecks related to cache usage [27], or to remove redundant memory loads [35].

Applying these methods on top of our memory tracing implementation could provide new insights into database systems and possibly reveal optimization opportunities that are hard to discover with current profiling approaches.

7 CONCLUSION

The state-of-the-art solution for identifying the root cause of performance problems related to memory accesses is to augment classical profiling with a memory trace. However, current approaches are not usable in practice due to their large runtime overhead.

In this work, we present an implementation for collecting memory traces via hardware-based sampling that leverages Intel's PEBS mechanism. In our experiments using the JCC-H benchmark, SAP HANA, and DuckDB, we illustrate for various use cases that memory traces enable us to analyze the runtime characteristics of a database system: The traces reveal access patterns of specific data structures, detect skew at byte-level, or allow us to estimate the working set size of a workload and to analyze the impact of table partitioning. In addition, we demonstrate that our implementation has a low runtime overhead—making it possible to collect memory traces in production environments.

ACKNOWLEDGMENTS

We thank Roman Dementiev and Michael Brendle for their helpful comments. The work has received funding from the Deutsche Forschungsgemeinschaft (DFG), Collaborative Research Center SFB 876, project C5 (`http://sfb876.tu-dortmund.de/`).

REFERENCES

[1] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017* (Washingon, DC, USA) *(ROSS '17)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. `https://doi.org/10.1145/3095770.3095773`

[2] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. 2008. HMTT: A Platform Independent Full-System Memory Trace Monitoring System. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Annapolis, MD, USA) *(SIGMETRICS '08)*. Association for Computing Machinery, New York, NY, USA, 229–240. `https://doi.org/10.1145/1375457.1375484`

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[4] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 103–119.

[5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85. `https://doi.org/10.1145/1409360.1409380`

[6] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology, USA. AAI0807735.

[7] Cisco. 2019. TPC Benchmark H Full Disclosure Report for Cisco UCS C480 M5 Rack-Mount Server using Microsoft SQL Server 2017 Enterprise Edition and Red Hat Enterprise Linux 7.6. `http://www.tpc.org/3337`

[8] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages*

and Operating Systems (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[9] S. Economo, D. Cingolani, A. Pellegrini, and F. Quaglia. 2016. Configurable and Efficient Memory Access Tracing via Selective Expression-Based x86 Binary Instrumentation. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 261–270. https://doi.org/10.1109/MASCOTS.2016.69

[10] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *Data Eng. Bull.* 35, 1 (2012), 28–33.

[11] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *Proc. VLDB Endow.* 5, 11 (July 2012), 1424–1435. https://doi.org/10.14778/2350229.2350258

[12] Intel. 2020. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/en-us/articles/intel-sdm.

[13] Intel. 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm.

[14] Intel. 2020. VTune Profiler. https://software.intel.com/vtune/.

[15] Intel. 2020. VTune Profiler: Memory Access Analysis. https://software.intel.com/en-us/vtune-help-memory-access-analysis.

[16] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 5.

[17] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *PVLDB* 5, 12 (2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[18] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *Proc. SIGMOD*. 1177–1184. https://doi.org/10.1145/1989323.1989448

[19] Letitia W. Li, Guillaume Duc, and Renaud Pacalet. 2015. Hardware-Assisted Memory Tracing on New SoCs Embedding FPGA Fabrics. In *Proceedings of the 31st Annual Computer Security Applications Conference* (Los Angeles, CA, USA) *(ACSAC 2015)*. Association for Computing Machinery, New York, NY, USA, 461–470. https://doi.org/10.1145/2818000.2818030

[20] Linux. 2020. perf. https://perf.wiki.kernel.org/.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[22] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-Directed Page Placement for CcNUMA via Hardware-Generated Memory Traces. *J. Parallel Distrib. Comput.* 70, 12 (Dec. 2010), 1204–1219. https://doi.org/10.1016/j.jpdc.2010.08.015

[23] Joe Mario. 2016. C2C – False Sharing Detection in Linux Perf. https://joemario.github.io/blog/2016/09/01/c2c-blog/.

[24] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[25] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[26] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. 2018. On the Applicability of PEBS Based Online Memory Access Tracking for Heterogeneous Memory Management at Scale. In *Proceedings of the Workshop on Memory Centric High Performance Computing* (Dallas, TX, USA) *(MCHPC'18)*. Association for Computing Machinery, New York, NY, USA, 50–57. https://doi.org/10.1145/3286475.3286477

[27] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 335–348. https://doi.org/10.1145/1755913.1755947

[28] Meikel Poess and Dmitry Potapov. 2003. Data Compression in Oracle. In *Proc. VLDB*. 937–947. http://dl.acm.org/citation.cfm?id=1315451.1315531

[29] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf

[30] Muhammad Aditya Sasongko, Milind Chabbi, Palwisha Akhtar, and Didem Unat. 2019. ComDetective: A Lightweight Communication Detection Tool for Threads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 21 pages. https://doi.org/10.1145/3295500.3356214

[31] Tobias Scheuer, Norman May, Alexander Böhm, and Daniel Scheibli. 2016. JexLog: A Sonar for the Abyss. *PVLDB* 9, 13 (2016), 1493–1496. https://doi.org/10.14778/3007263.3007292

[32] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. 2015. Low-Overhead Detection of Memory Access Patterns and Their Time Evolution. In *Euro-Par 2015: Parallel Processing*, Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–69.

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

[34] Sam Silvestro, Hongyu Liu, Tong Zhang, Changhee Jung, Dongyoon Lee, and Tongping Liu. 2018. Sampler: PMU-Based Sampling to Detect Memory Errors Latent in Production Software. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 231–244. https://doi.org/10.1109/MICRO.2018.00027

[35] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant Loads: A Software Inefficiency Indicator. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 982–993. https://doi.org/10.1109/ICSE.2019.00103

[36] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. 2009. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. *SIGPLAN Not.* 44, 3 (March 2009), 121–132. https://doi.org/10.1145/1508284.1508259

[37] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. 2014. ADDICT: Advanced Instruction Chasing for Transactions. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1893–1904. https://doi.org/10.14778/2733085.2733095

[38] Pundefinednar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in Wonderland: Where Do Cache Misses Come from in Major OLTP Components?. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware* (New York, New York) *(DaMoN '13)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/2485278.2485286

[39] Transaction Processing Performance Council (TPC). 2018. *TPC Benchmark H (Decision Support) Standard Specification.* Technical Report. http://www.tpc.org/tpch/

[40] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. 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, 1009–1024. https://doi.org/10.1145/3035918.3064029

[41] Vince Weaver. 2020. Linux Programmer's Manual – perf_event_open. http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[42] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 13, 15 pages. https://doi.org/10.1145/3190508.3190511