Large-Scale In-Memory Analytics on Intel® Optane™ DC Persistent Memory

Anil Shanbhag
MIT
anil@csail.mit.edu

Nesime Tatbul
Intel Labs and MIT
tatbul@csail.mit.edu

David Cohen
Intel
david.e.cohen@intel.com

Samuel Madden
MIT
madden@csail.mit.edu

ABSTRACT

New data storage technologies such as the recently introduced Intel® Optane™ DC Persistent Memory Module (PMM) offer exciting opportunities for optimizing the query processing performance of database workloads. In particular, the unique combination of low latency, byte-addressability, persistence, and large capacity make persistent memory (PMem) an attractive alternative along with DRAM and SSDs. Exploring the performance characteristics of this new medium is the first critical step in understanding how it will impact the design and performance of database systems. In this paper, we present one of the first experimental studies on characterizing Intel® Optane™ DC PMM's performance behavior in the context of analytical database workloads. First, we analyze basic access patterns common in such workloads, such as sequential, selective, and random reads as well as the complete Star Schema Benchmark, comparing standalone DRAM- and PMem-based implementations. Then we extend our analysis to join algorithms over larger datasets, which require using DRAM and PMem in a hybrid fashion while paying special attention to the read-write asymmetry of PMem. Our study reveals interesting performance tradeoffs that can help guide the design of next-generation OLAP systems in presence of persistent memory in the storage hierarchy.

ACM Reference Format:
Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-Scale In-Memory Analytics on Intel® Optane™ DC Persistent Memory. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3399666.3399933

1 INTRODUCTION

Hardware trends have greatly influenced the development of data management systems. Historically, most of the data was stored on (rotating) disks, and only small fractions of the data would be kept in main memory (in a buffer pool). The increase in DRAM card capacities along with the increase in the number of DIMM slots per socket allowed machines to have up to small terabytes of DRAM. This change allowed systems to keep large fractions / all of their data directly in memory. Main memory is more than an order of magnitude faster than disk and allows random access. Thus, in comparison to disk-based systems, in-memory database systems offer significant performance improvements. A number of in-memory data systems were proposed over the years [9, 12, 16]. These systems make use of DRAM-only storage and offer outstanding performance, but tend to fail or degrade heavily if the data does not fit into main memory.

Current hardware trends have cast strong doubt on the viability of pure in-memory systems. DRAM sizes are not increasing significantly any more. Ten years ago, one could conceivably buy a commodity server with 1TB of memory for a reasonable price. Today, affordable main memory sizes might have increased to 2TB, but going beyond that disproportionately increases the costs. This limits scalability. Further, these large main memory servers are multi-socket, which introduces complexity in terms of handling remote memory accesses (NUMA).

Over the past decade, memory technologies such as byte-addressable persistent memory (PMem) (also called non-volatile memory (NVM)) devices have been developed [3, 20]. Although they are slower, they have higher densities than DRAM. The Intel® Optane™ DC Persistent Memory Module (PMM) is the first such product on the market [1]. The Intel® Optane™ DC PMM is plugged directly into the memory bus via traditional DIMM slots. It has 16 times higher density (512GB capacity per card vs. 32GB DRAM cards used in servers) and is cheaper per GB [14].

In this paper, we take a close look at using Intel® Optane™ DC PMMs in the context of analytical database workloads. We evaluate the performance of three basic access patterns: sequential, selective, and random, on both PMem and DRAM, and explain why the runtime ratio would differ from the PMem to DRAM bandwidth ratio. We present an implementation of the Star Schema Benchmark (SSB) [17], and compare its performance with data starting in PMem vs. data starting in DRAM, as well as against a state-of-the-art DRAM-based in-memory OLAP DBMS. We further show that using a hybrid system with DRAM and PMem can allow us to run much larger workloads efficiently with minimal degradation in performance in contrast to a purely DRAM-based, in-memory system when the query intermediates fit in DRAM.

In a DRAM-based in-memory system, the memory is used both for storing the data and as a scratch space during query execution. In a hybrid system, data can be stored and accessed directly from PMem, thereby freeing DRAM for query intermediates. Hence, the only situations where we run out of main memory are when the intermediates are larger than size of DRAM, like when joining two large tables. PMem is unique in that there is read-write asymmetry - the read throughput is significantly higher than the write throughput. In this context, we evaluate a number of join algorithms and analyze their performance on the hybrid system. We show that a previous state-of-the-art join algorithm, called Segmented Grace Hash Join (SGHJ) [25], devolves into a Nested Loops Join (NLJ) when applicable. We propose simple rules to decide the join algorithm to use when build-phase hash table does not fit in DRAM.

In summary, we make the following contributions:

- We present an analysis of basic access patterns seen in analytical workloads on DRAM and PMem.
- We present an implementation of SSB and show that, while PMem read bandwidth is 3x lower than DRAM read bandwidth, on-average, the workload is only 1.6x slower when data is in PMem vs. when data is in DRAM.
- We present a detailed evaluation of larger-than-DRAM algorithms for running joins on a hybrid DRAM-PMem system.

Overall, we believe our results can offer an insightful guide to implementors as to what sorts of basic performance behaviors can be expected when running OLAP workloads in the hybrid setting with Intel® Optane™ DC PMM.

2 BACKGROUND

2.1 Persistent Memory

Persistent memory (PMem), also called non-volatile memory (NVM), is a class of memory technologies that combines the

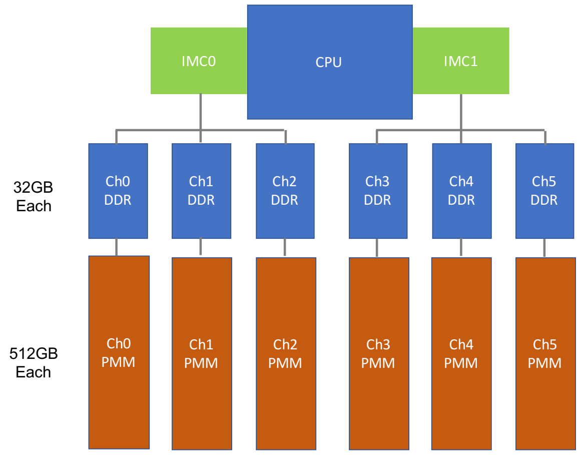


Figure 1: Schematic of a Hybrid DRAM+PMem System

low latency and byte-addressability of DRAM with the persistence and large capacity of SSD. The first commercially available PMem is the one by Intel called the Intel® Optane™ DC Persistent Memory Module (PMM) [1]. PMM comes in DIMM form factor and embeds up to 512GB storage capacity, i.e., double the capacity of the largest DIMMs available (256GB) and order of magnitude higher than 32GB DRAM DIMMs used in servers today. Figure 1 shows a typical system configuration of the hybrid system on a single node with DRAM and PMM. Each socket has 12 DIMM slots. 6 DIMM slots are occupied by 32GB DDR4 DRAM modules and the other 6 slots are occupied by 512GB PMMs. That totals to 192GB of DRAM and 3TB of PMem per socket. The processor communicates with DRAM using the standard DDR4 protocol, whereas for PMM it uses the DDR-T protocol, both happening via the Integrated Memory Controller (IMC). PMMs use 256B cache line access granularity, which is larger than the 64B cache line access granularity used in DRAM. The PMM has an on-DIMM Apache Pass controller that translates 64B load/stores into 256B accesses.

The PMM operates in three different modes:

- **Memory Mode**: PMMs act as volatile, byte-addressable main memory. DRAM acts as a cache for PMM and is not visible to the user.
- **App Direct Mode**: PMMs act as persistent storage separate from the primary memory hierarchy.
- **Mixed Mode**: Part of PMMs is used as main memory and the remaining part is used as persistent storage. DRAM acts as cache for PMM.

The advantage of the Memory Mode is that it works transparently for legacy software. The DRAM serves as an additional hardware-managed caching layer ("L4 cache"). However, it does not provide fine-grained control over where the data resides, and past work has shown that Memory Mode adds around 10% overhead compared to App Direct

Mode [24]. For this reason, we focus on using App Direct Mode in the remainder of this paper. When we use PMem in the App Direct Mode, the memory appears as a block storage device. User can allocate memory on the device by creating a file and map the file into address space as follows:

```
col = pmem_map_file("/mnt/pmem12/col", col_size,
  PMEM_FILE_CREATE, 0666, &len, &is_pmem);
```

The resulting pointer can be used like an array in memory. Note that accesses to `col` bypass the OS block cache.

2.2 Related Work

Researchers have worked on adapting OLTP databases to use PMem. Chen et al. proposed the wB-Tree [10], an NVM-based persistent B+-Tree that keeps sorted in-direction arrays to mitigate the fact that tree nodes are kept unsorted. Furthermore, Yang et al. proposed the NV-Tree [26], a cache-conscious B-Tree that does not enforce the consistency of the inner nodes since they can be rebuilt from the leaf nodes, thereby reducing the number of PMem writes. Other NVM-based structures that build on the above techniques and further optimize failure-atomicity handling have been proposed, e.g., the FPTree [18], the Write-Optimal Adaptive Radix Tree [15], and the Bz-Tree [4]. Arulraj et al. proposed to adapt the cost model of query optimizers to take into account PMem's read/write asymmetry [5]. In-memory OLAP workloads are largely read-only. Van Ren et al. proposed using PMem as a part of three-tiered hierarchy for OLTP workloads [23]. Psaropoulos et al. worked on hiding the latency difference between PMem and DRAM for database applications by interleaving the execution of parallel work in index joins and tuple reconstruction using coroutines [19]. These techniques are only effective on single-threaded processing; we largely focus on running in parallel using all threads, at which point queries are bandwidth-bound and not latency-bound.

Andrei et al. present architectural choices involved in integrating NVM into the SAP HANA in-memory DBMS [2]. Their work focuses on storage design to support OLTP and OLAP workloads, whereas in this paper we focus on the performance aspect of using the hybrid system. Van Renen et al. performed performance evaluations of PMM in terms of bandwidth and latency [24], and developed guidelines for efficient usage of PMM and tuned I/O primitives, namely log writing and block flushing. We build on the insights of this to characterize performance on OLAP workloads. PMem is unique in that there is read-write asymmetry - read throughput is significantly higher than write throughput. This requires a redesign of algorithms to reduce the number of writes to durable storage. Viglas proposed the Segmented Grace Hash Join (SGHJ) [25], which, unlike the regular Grace Hash Join (GHJ) [13], materializes only a fraction of the input partitions, and continuously iterates over the rest of the input to process the remaining partitions. The associated read amplification does not hurt performance given the read-write asymmetry. We revisit this work and show that SGHJ devolves into either GHJ or read-only Nested Loops Join (NLJ) based on the size of the build relation and memory buffer size.

3 ANALYTICS ON PMEM

In this section, we evaluate the performance of running analytical workloads on a hybrid system with DRAM and PMem. We compare the performance of key access patterns and a SQL query benchmark with data stored on DRAM vs. with data stored on PMem. In this section, we restrict ourselves to the case where the dataset fits in DRAM. A key benefit of using PMem is the ability to process datasets much larger than size of DRAM. In the next section, we examine the case when data does not fit in DRAM.

We start by comparing the performance of DRAM vs. PMem on three key access patterns seen in analytical workloads:

- Sequential Access (Read and Write): Associated with accessing the columns before any selections.
- Selective Access (Read): Associated with accessing a column after a selection or selective join.
- Random Access (Hash Probe): Associated with joins.

Then, we evaluate the performance of the Star Schema Benchmark on a hybrid system and contrast it with a purely in-DRAM solution.

For the experiments, we are using a two-socket, 2nd Generation Intel® Xeon® Scalable Processor (Cascade Lake) system with 24 physical (48 virtual) cores on each node. Each socket's memory configuration is as shown in Figure 1. All the experiments were run on a single socket using all 48 virtual cores unless otherwise stated. The machine is running Fedora with a Linux kernel version 4.15.6. All results are reported as an average of 3 runs.

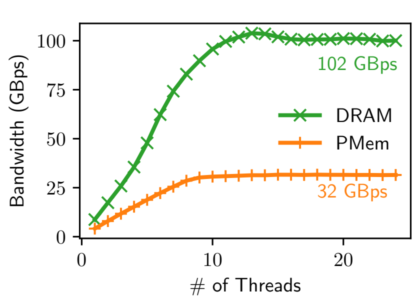
3.1 Performance of Basic Access Patterns

In this section, we look at three common access patterns that occur in analytical workloads run in an in-memory column-store to evaluate how their performance changes with PMem.

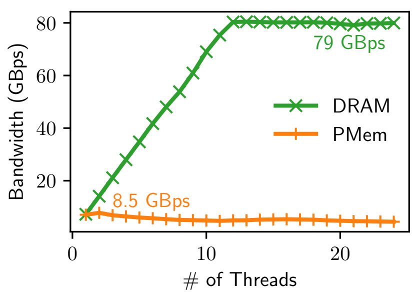
Sequential Access

To compare performance on sequential accesses, we measure the read and write bandwidth of DRAM and PMem using streaming read and streaming write, respectively, with varying number of threads. Note that each thread runs on a separate physical core.

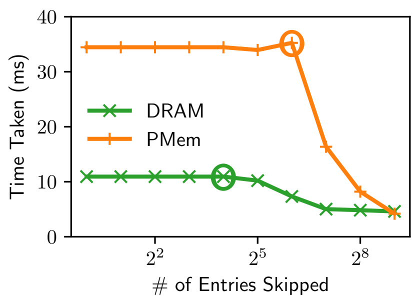
Figure 2a shows the results for sequential read. The single-thread read bandwidth for DRAM and PMem is 8.7 GBps and 4.1 GBps, respectively. We note that twelve threads are sufficient to saturate bandwidth on both DRAM and PMem.



(a) Sequential Read



(b) Sequential Write



(c) Selective Read

Figure 2: Performance Behavior for Sequential and Selective Access Patterns

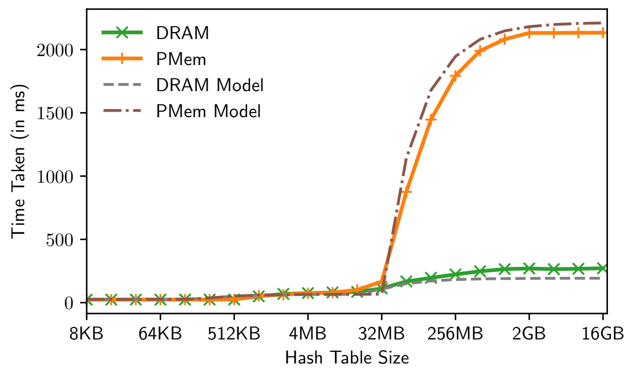


Figure 3: Hash Probe Performance (Random Access Pattern)

The peak read bandwidth of DRAM and PMem is 102 GBps and 32 GBps, respectively. Hence, PMem is 3.2x slower for reads.

Figure 2b shows the results for sequential write. The single-thread write bandwidth for DRAM and PMem is 7.2 GBps and 7 GBps, respectively. The peak write bandwidth of DRAM is 79GBps. For PMem, the write bandwidth peaks at 8.5 GBps with two threads. Using additional threads decreases the bandwidth achieved.

We define λ to be the ratio of the maximum read bandwidth to the maximum write bandwidth for the medium. Using the bandwidth numbers, we get λ_{DRAM} is 1.3 and λ_{PMem} is 3.8 — there is significant asymmetry in the read and write bandwidth of PMem.

Selective Access

In a query with multiple selection predicates, after the first predicate is evaluated, entries in subsequent columns with predicates are selectively accessed based on whether they passed all previous predicates. To mimic this access pattern, we scan an array of 500 million 4-byte integers and selectively access every j^{th} entry, where j is the number of entries skipped. Figure 2c shows the results. Time taken with array on PMem drops when skip size is greater than 256B (which is the granularity of accesses to PMem) compared to 64B for DRAM.

Random Access

Hash Join (HJ) is the most popular join algorithm for in-memory DBMS. Hash Join runtimes are typically dominated by the probe phase, which involves random access into a hash table in memory. To compare performance of DRAM vs. PMem on random access, we measure the runtime of the probe phase of the following join query:

```
SELECT SUM(A.v + B.v) AS checksum
FROM A, B
WHERE A.k = B.k
```

where each table, A and B, consists of two 4-byte integer columns, k, v. The two tables are joined on key k. We keep the size of the probe table fixed at 256 million tuples, totaling 2 GB of raw data. We use a hash table with 50% fill rate. We vary the size of the build table such that it produces a hash table of the desired size in the experiment. The hash table resides on DRAM or PMem depending on the test. The microbenchmark is the same as what past works use [6–8, 21].

Figure 3 shows the results. When the hash table fits in the L2/L3 cache, the performance is the same for both memories, as the performance is bound by cache bandwidth and not by where data starts out. Once hash table size is bigger than L3 cache, we end up doing random access into the hash table stored in respective memories. The access granularity of PMem is 256B while for DRAM is 64B - a factor of 4x. Further, DRAM read bandwidth is 3.2x higher than that of PMem. Hence, one would expect at least 12x difference in performance. However, we observe that, when the hash table size is 16GB, the ratio of DRAM to PMem runtime is 8x.

To understand the reason for this, we plot the theoretical runtimes that would be achieved assuming memory bandwidth is saturated. If the hash table size is larger than the size

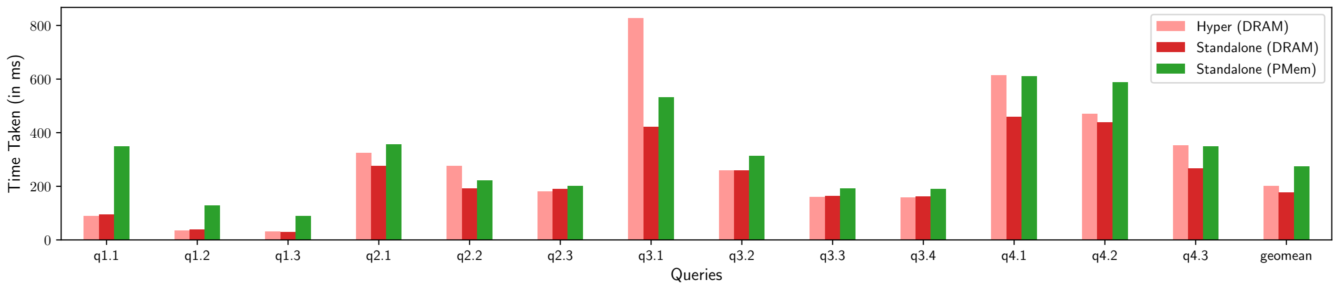


Figure 4: Star Schema Benchmark Performance

of the last level cache, we expect the runtime to be:

$$runtime = \frac{4 \times 2 \times |P|}{B_p} + (1-\pi)(\frac{|P| \times C}{B_h}) \quad (1)$$

where $|P|$ is the cardinality of the probe table, B_p and B_h are the read bandwidths of the memories on which the probe table and hash table reside respectively, C is the cache line size accessed on probe, and π is the probability of an access hitting the last level cache. The first term is the time taken to scan the probe table from the respective memory. The second term is the time for probing the hash table. Note that each probe accesses an entire cache line. A similar model can be used when hash table is smaller than size of last level cache. We refer the interested reader to our complete cost model published in an earlier study [22]. The runtimes based on the model are plotted as `DRAM Model` and `PMem Model` for DRAM and PMem, respectively.

`PMem Model` closely tracks the actual performance with an error of 3% at the plateau. `DRAM Model`, on the other hand, is 30% less than the actual runtime. This is due to memory controller not being able to fulfill requests at DRAM memory bandwidth speed, hence the memory stalls. In conclusion, random accesses to PMem are 8x slower than to DRAM.

3.2 Star Schema Benchmark Performance

Now that we have a good understanding of how different access patterns behave on both DRAM and PMem, we will evaluate the performance of a workload of full SQL queries on both DRAM and PMem. For the workload, we use the Star Schema Benchmark (SSB) [17]. SSB is a simplified version of the more popular TPC-H benchmark. It has one fact table *lineorder*, and four dimension tables *date, supplier, customer, part*, which are organized in a star schema fashion. There are a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments, we first run the benchmark with a scale factor of 100, which will generate the fact table with 120 million tuples. The total dataset size is around 70GB.

We implement the benchmark queries in C++ and compare the performance of queries with data starting out in PMem (`Standalone (PMem)`) to performance with data starting out in DRAM (`Standalone (DRAM)`). As a sanity check, we also compare against Hyper (`Hyper (DRAM)`), a state-of-the-art in-memory OLAP DBMS. Note that the Hyper runtimes exclude the query compilation time.

Figure 4 shows the results. The queries in the Star Schema Benchmark can be broken into two sets: 1) the query flight $q1.x$ consists of queries with selections directly on the fact table with no joins and 2) the query flights $q2.x$, $q3.x$, $q4.x$ consist of queries with no selections on fact table and multiple joins — some of which are selective. Comparing `Standalone (DRAM)` to `Standalone (PMem)` on set 1, queries with `Standalone (PMem)` are on average 3x slower than `Standalone (DRAM)`. Query runtimes in set 1 is dominated by sequential and skipped access patterns. Hence their performance ratio is close to the bandwidth ratio of the two memories.

Comparing the performance of the two memories on set 2, we see that queries in `Standalone (PMem)` are on average 1.4x slower than in `Standalone (DRAM)`. In this set, in each query, the fact table joins against multiple dimensions tables. The query runtime is dominated by random access patterns involved in the hash probing. Revisiting Eq. 1, we have the probe table residing on PMem / DRAM, while the hash table fits in DRAM — as a result, change in memory only increases the first term, while the second term remains the same for both.

In aggregate, the geometric mean of runtimes of `Standalone (PMem)` is 1.6x higher than that of `Standalone (DRAM)`. Hence, the performance hit from having data stored in PMem is less than the bandwidth ratio when the intermediates fit in DRAM. In most commonly used workloads, the dimension tables are significantly smaller than fact tables; further, hash table is built on select columns and not the entire table.

Finally, comparing `Standalone (DRAM)` to `Hyper (DRAM)` shows that the former does on an average 1.17x better than the latter. Hyper is likely using different underlying implementation for certain operators. The comparison serves as a sanity check and shows that our implementation is competitive compared to a state-of-the-art OLAP DBMS.

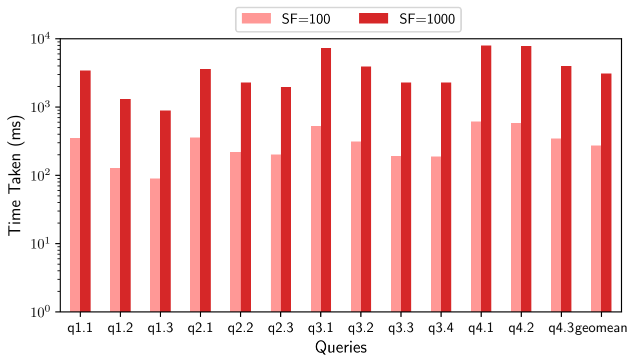


Figure 5: Star Schema Benchmark Performance on PMem with Scale Factor SF=1000 vs. SF=100

4 HANDLING LARGER DATASETS

In the previous section, we analyzed the performance of an analytical workload in the hybrid system, when the dataset and query intermediates fit in DRAM. In a DRAM-based in-memory system, DRAM is used both for storing the data and and as scratch space during query execution. In the hybrid system, data resides on PMem. Hence, when handling larger datasets, we have two scenarios: 1) when the query intermediates fit in DRAM, and 2) when the intermediates are larger than size of DRAM, like when joining two large tables.

4.1 Intermediates Fit in DRAM

In data warehouses modeled on a star schema, there are one or more fact tables referencing any number of dimension tables. The dimension tables are usually much smaller than the fact tables. A query with Hash Joins ends up building hash tables on the dimension tables. Hence, while the database size may be larger than size of DRAM, it is possible to process queries on this database using PMem with minimal changes, as the query intermediates built during query execution fit in DRAM. To show the performance characteristics of running a workload where intermediates fit in DRAM, we run the Star Schema Benchmark queries using `Standalone (PMem)` with a scale factor of 1000 vs. with a scale factor of 100. In the former, the fact table size is larger than size of DRAM.

Figure 5 shows the results. Note that the runtimes are plotted on a log scale. The average ratio of query runtimes is 11.4x and ratio of the geometric mean of the runtimes is 11.3x. This is close to ratio of data sizes which is 10x. Hence, the performance scales linearly with the data size when intermediates fit in DRAM.

4.2 Intermediates Larger than DRAM

A common example for having intermediates larger than the DRAM size is when joining two large tables. In this section, we discuss different join algorithms that can be used when the hash table does not fit in DRAM and benchmark their performance.

Grace Hash Join (GHJ)

Grace Hash Join (GHJ) is a commonly used join algorithm in disk-based DBMS [11]. Given inputs T and V with $|T| \leq |V|$ and a memory budget of M ($M > \sqrt{|T|}$, i.e., GHJ is applicable), GHJ would partition each side into $k = \lceil |T|/M \rceil$ partitions such that the tuples in the i^{th} partition of T join only with the tuples in the i^{th} partition of V. In the next step, each partition pair is joined one at a time.

Segmented Grace Hash Join (SGHJ)

PMem is unique in that there is read-write asymmetry - the read throughput is significantly higher than the write throughput. GHJ involves writing to PMem; an alternative is to use the Nested Loops Join (NLJ), which involves no writes. If λ is the read to write bandwidth ratio of the medium, depending on the size of input relations and λ, one could choose NLJ or GHJ.

Viglas studied this problem and presented a number of hybrid join algorithms that adapt to the read-write asymmetry λ, out of which the Segmented Grace Hash Join (SGHJ) was found to be the best performing algorithm [25]. In SGHJ, given a number of partitions $k = \lceil |T|/M \rceil$, we choose to materialize only some number, x, of them, and continuously iterate over the rest of the inputs to process the remaining $k-x$ partitions.

The total cost, $J(x)$, of the algorithm is given by Eq. 2, where r denotes the read cost per tuple. We scan the input to extract the x partitions; we offload these partitions; and then read them back to process their partial join. We therefore fully scan T and V, write $x(|T|+|V|)/k$ buffers, and next read back the $x(|T|+|V|)/k$ buffers one pair at a time to process the partial joins, where $|T|/k$ (resp. $|V|/k$) is the size of each partition of T (resp. V). The cost of this step is represented by the first two factors in Eq. 2. In the next step, we iterate over both inputs $k-x$ times, each time processing one partition. The cost of iterating over both inputs $k-x$ times to process the remaining partitions is $r(k-x)(|T|+|V|)$.

$$J(x) = r(|T|+|V|) + rx(1+\lambda)(\frac{|T|+|V|}{k}) + r(k-x)(|T|+|V|) \tag{2}$$

The cost is parameterized by x, the number of partitions that will be written. x is a measure of the write-intensity of the algorithm. Simplifying the above:

$$J(x) = r(|T|+|V|)(1+(\lambda+1)x/k+(k-x))$$

The cost of GHJ is $r(|T|+|V|)(\lambda+2)$. We can use SGHJ if it costs less than that of GHJ. Here we found that a mistake was made in the formulation of Viglas [25], which led to a wrong conclusion that the choice of x is based on a non-linear function of k and λ. Correctly comparing the two costs, we

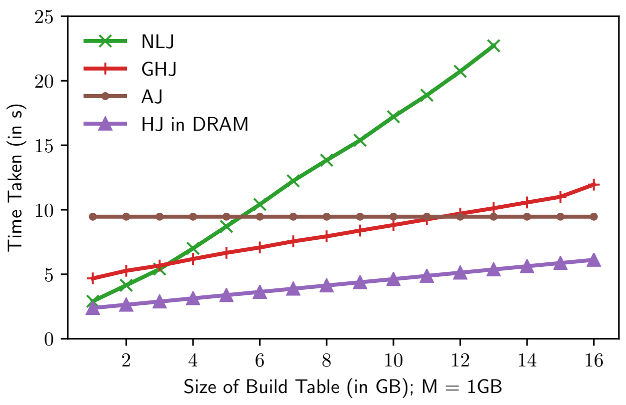


Figure 6: Join Performance for Large Datasets

find that one can use SGHJ if:

$$
\begin{aligned}
(1+(\lambda+1)x/k+(k-x)) &< \lambda+2 \\
k(1-x/k) &< (1-x/k)(\lambda+1) \\
k &< \lambda+1
\end{aligned}
$$

Hence, we see that SGHJ is faster, when $k<\lambda+1$. The cost is parameterized by x and the cost function can be rewritten as:

$$J(x)=r(|T|+|V|)(1+k+((\lambda+1)/k-1)x)$$

Since we know $k<\lambda+1$, the cost is directly proportional to x and minimized at $x=0$. Hence, the SGHJ, when applicable (i.e., $k<\lambda+1$), defaults to using the NLJ.

Array Join (AJ)

A key difference between PMem and SSD is that PMem is byte-addressable and supports random access. In column stores, in many occasions, the dimension tables are already sorted and their keys are indexed (e.g., 0 to n). Hence in this case we can do the join with the table in PMem by scanning the probe side, for each table accessing the column entry directly from PMem. This removes the need to build the hash table.

Experimental Evaluation

To join tables where the build side is bigger than the size of DRAM, we have so far discussed three algorithms: GHJ, NLJ, and AJ, where AJ is applicable only in specific scenarios (and SGHJ was shown to default to NLJ). To evaluate their performance, we use the same join benchmark used to evaluate the random access pattern in Section 3.1. We keep the size of the probe table fixed at 4 billion tuples, totaling around 16GB. We keep the memory budget fixed at $M=1$GB and vary the size of build side from 128 million (1GB) to 4 billion tuples (16GB). The key column of the build side is Knuth-shuffled, except when evaluating AJ, in which case, it is ordered 1 to n. For comparison, we also show the performance when the hash join has sufficient memory to store the entire hash table in DRAM (`HJ in DRAM`).

Figure 6 shows our results. As expected, `HJ in DRAM` is the fastest. Let $k=\lceil |T|/M \rceil$ where T is the build side. For small values of k, NLJ does better than GHJ. For PMem, $\lambda=3.8$; based on our previous estimate we expect the transition point at $k=\lambda+1$, which is 4.8. However, in the experiment the transition happens when $k>3$. This is because we ignored the DRAM access cost in the equation previously, which decreases the transition point.

The time taken in AJ is a constant, as it depends on the size of the probe side, which is a constant in this microbenchmark. AJ is the best algorithm when $k \geq 12$. Note that while PMem supports random access, the granularity of PMem accesses is 256B compared to 64B on DRAM. When $k \geq 12$, the size of the build table is comparable to the size of the probe table and the cost of building the partitions in GHJ is higher than cost of 256B random accesses to PMem. If the build side is not already indexed, one would be tempted to build a hash table in PMem and then use AJ. Building a hash table on the build side in PMem involves two write passes over the build side and algorithm performs strictly worse than GHJ.

5 CONCLUSION

This paper analyzed the performance characteristics of running analytical workloads on a hybrid system with Persistent Memory and DRAM. We evaluated the difference in performance on basic access patterns like hash probes and use performance models to explain them. Our analysis on SSB, a popular analytics benchmark, shows that using PMem to store data leads to only 1.6x slowdown compared to purely an in-DRAM system, while giving us an order of magnitude higher data storage capacity. We discussed join algorithms to handle the case when join hash tables do not fit in DRAM, and presented simple rules to choose the right algorithm. Our results should help system implementors effectively evaluate the tradeoffs of using PMem when designing their systems.

ACKNOWLEDGMENTS

This research is supported by Intel Corporation as part of the MIT Data Systems and AI Lab (DSAIL). We thank Sanjay Kumar and Karthik Sukumar from Intel Labs for their guidance and technical help with using PMem.

REFERENCES

[1] Intel® Optane™ DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory/.

[2] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, et al. Sap hana adoption of non-volatile memory. *Proceedings of the VLDB Endowment*, 10(12):1754–1765, 2017.

[3] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):1–35, 2013.

[4] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[5] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758, 2017.

[6] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[7] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.

[9] P. A. Boncz et al. *Monet: A next-generation DBMS kernel for query-intensive applications*. Universiteit van Amsterdam [Host], 2002.

[10] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[11] D. J. DeWitt and R. Gerber. *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1985.

[12] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[13] J. R. Goodman. An investigation of multiprocessor structures and algorithms for data base management. 1982.

[14] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel® Optane™ Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[15] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. {WORT}: Write optimal radix tree for persistent memory storage systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 257–270, 2017.

[16] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[17] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[18] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. 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*, pages 371–386, 2016.

[19] G. Psaropoulos, I. Oukid, T. Legler, N. May, and A. Ailamaki. 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*, pages 1–8, 2019.

[20] S. Raoux, F. Xiong, M. Wuttig, and E. Pop. Phase change materials and phase change memory. *MRS bulletin*, 39(8):703–710, 2014.

[21] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1961–1976. ACM, 2016.

[22] A. Shanbhag, X. Yu, and S. Madden. A study of the fundamental performance charecteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. ACM, 2020.

[23] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing Non-Volatile Memory in Database Systems. In *ACM SIGMOD Conference*, pages 1541–1555, 2018.

[24] A. Van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7, 2019.

[25] S. D. Viglas. Write-limited sorts and joins for persistent memory. *Proceedings of the VLDB Endowment*, 7(5):413–424, 2014.

[26] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 167–181, 2015.