High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance

Maximilian Kuschewski
maximilian.kuschewski@tum.de
Technische Universität München

Jana Giceva
jana.giceva@tum.de
Technische Universität München

Thomas Neumann
neumann@in.tum.de
Technische Universität München

Viktor Leis
leis@in.tum.de
Technische Universität München

ABSTRACT

This paper aims to bridge the gap between fast in-memory query engines and slow but robust engines that can utilize external storage. We find that current systems have to choose between fast in-memory operators and slower out-of-memory operators. We present a solution that leverages two independent but complementary techniques: First, we propose adaptive materialization, which can turn any hash-based in-memory operator into an out-of-memory operator without reducing in-memory performance. Second, we introduce self-regulating compression, which optimizes the throughput of spilling operators based on the current workload and available hardware. We evaluate these techniques using the prototype query engine Spilly, which matches the performance of state-of-the-art in-memory systems, but also efficiently executes large out-of-memory workloads by spilling to NVMe arrays.

ACM Reference Format:
Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance . In *Proceedings of SIGMOD 2025*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Changing hardware drives system evolution. Database systems depend on a plethora of abstractions which allow them to evolve with the ever-changing hardware landscape. Relational algebra and SQL, for example, have allowed databases to adapt their internals to new hardware while keeping a mostly consistent interface across half a century. Some abstractions, however, turned out to be sorely lacking: Many classical database components, for instance, became major performance bottlenecks as hardware evolved [20, 41, 75]. This realization sparked a long line of work on efficient in-memory engines [24–26, 45, 46, 50, 51, 57, 61, 64, 66, 70, 71, 82] that improved performance by orders of magnitude. Thus, systems research needs to constantly re-evaluate the validity of assumptions and the usefulness of abstractions in the face of new hardware.

The rapid development of NVMe SSDs. Of the many recent hardware developments, one stands out in particular: In just five years, NVMe SSD bandwidth has increased twentyfold while storage prices reduced threefold, as Figure 1 shows. An array of PCIe 5.0 NVMe SSDs can now achieve more than 100 GB/s read throughput while costing a fraction of the equivalent DRAM capacity. We need to evaluate whether our systems and their abstractions are capable of exploiting this disruptive development.

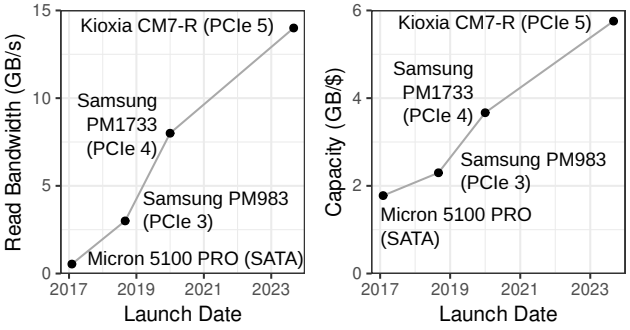

Figure 1: SSD read bandwidth and cost evolution [12–15].

How fast SSDs impact state-of-the-art systems. The emergence of high-throughput NVMe SSDs affects current engines in two ways: First, pure in-memory engines become less economically viable as they rely exclusively on expensive DRAM and are not equipped to read from or spill to much cheaper SSDs. Second, systems capable for out-of-memory processing can use, but not fully utilize, modern NVMe SSDs because they were designed with hard disks in mind. This leaves users with a trade-off: Use a fast in-memory system that fails when data grows too large; or use a robust out-of-memory-capable system that is significantly slower. It also begs the question: Why are there two categories of systems at all?

In-memory vs. out-of-memory systems: The missing link. When executing a query, systems face a trade-off: They need to decide whether to apply fast-in memory operators or use robust out-of-memory-capable operators that are slower – based on unreliable cardinality estimates [52], and for each operator in each query. For instance, in-memory systems can use a fast in-memory hash join where out-of-memory-capable systems need to use a slower grace or hybrid hash join to allow spilling. This paper proposes eliminating this trade-off by introducing a *common abstraction* that transparently handles materialization, larger-than-memory intermediary results, and spilling – while also staying fast in-memory.

Umami: Unifying in-memory materialization and spilling. We propose *Umami* (Unified Materialization Management Interface), which allows unifying in-memory operators with spilling operators, obviating the need for physical operator choice. Umami also enables hardware-aware spilling to make systems more resilient to hardware evolution. Umami leverages two independent but complementary parts that constitute this paper's first two **contributions**:

SIGMOD 2025, July 2025, Berlin, Germany

(1) Umami uses ***adaptive materialization***, which allows hash-based in-memory operators to start partitioning and spilling dynamically at runtime (Section 4.2).
(2) Umami applies ***self-regulating compression***, which optimizes the throughput of spilling operators based on the available CPU and I/O bandwidth (Section 4.4).

As a third contribution, we design operators based on Umami:

(3) We propose ***unified operators*** that are as fast as pure in-memory operators but can switch to efficient out-of-memory processing using modern NVMe SSDs (Sections 4.5 and 4.6).

Evaluating Umami on NVMe arrays. We validate this approach using a prototype query engine based on Umami that is capable of executing TPC-H. This engine, called Spilly, serves as a platform for designing and evaluating Umami-based aggregation and join operators. Spilly achieves in-memory performance comparable to the state-of-the-art in-memory engines Hyper [45] and DuckDB [69], but can also execute large out-of-memory workloads. To our knowledge, Spilly is the only query engine that can effectively utilize multiple modern PCIe 5.0 NVMe SSDs. As demonstrated in Section 6, Spilly can execute TPC-H scale factor 10,000 (10 TB of data) on a system with only 384 GB main memory while maintaining 86% of in-memory tuple throughput and spilling 5.5 TB of data.

2 RELATED WORK

Scanning from SSD. Scanning data stored on SSDs efficiently is an important prerequisite for SSD-based query processing. A recent paper [78] equips the scan and index lookup operators of the DBMS Umbra [60] with asynchronous I/O and reports large performance improvements over the synchronous implementation. Other recent work [62, 63] re-evaluates in-memory caching in the presence of fast NVMe storage and develops new caching policies that are better suited for high-throughput storage and heterogeneous hardware.
Spilling to SSD. In contrast to scanning, there is limited research that addresses efficient spilling to SSDs. Recent work [44] provides an in-depth analysis of the hybrid hash join (HHJ) operator, including experiments that spill to SSD. The study focuses on algorithmic considerations, thus centering on single-threaded experiments that process gigabytes of data with tens of megabytes per second on small servers. However, modern NVMe arrays can read and write with tens of gigabytes per second, approaching even the processing speed of in-memory systems. In this work, we propose techniques that allow such state-of-the-art engines to exploit high-throughput NVMe arrays without sacrificing their in-memory performance.
External aggregation in DuckDB. Recent work by Kuiper et al. [48] also recognizes the need for operators that gracefully switch from in-memory to out-of-memory processing. The paper focuses on memory management and argues for a unified buffer manager, which manages both table pages and temporary data. Experiments, executed on hardware comparable to a modern laptop, show the aggregation operator scaling robustly beyond main memory. Umami, in contrast, focuses less on memory management, and more on efficient spilling to multiple high-throughput SSDs, and can be applied to any hash-based operator. Section 4.6 details further differences between Spilly's and DuckDB's aggregation operator.
Older work on SSD-based OLAP. While recent work on OLAP using NVMe SSDs is scarce, SSDs garnered some attention when they became widely available. Most of this work focused on the transition from disks to SSDs. At that time, SSDs were much more expensive [29], so prior work usually argued for a multi-layer hierarchy where SSDs act as a cache for disks [21, 30, 31, 35, 37, 38, 47, 72, 76].
SSD-based OLTP. OLTP on SSDs has received more attention than OLAP, as evidenced by systems such as RocksDB [4], ScyllaDB [9], MosaicDB [43], and LeanStore [53]. Recent work [40] on LeanStore demonstrates how a transactional system can fully exploit an array of PCIe 4.0 SSDs, using a setup that is similar to ours.
Hardware cost. Our measurement server (c.f., Section 6.1) uses an array of eight 3.84 TB PCIe 5.0 SSDs. This setup strikes a balance between capacity and bandwidth, and is not much more expensive than older SSD generations, as the following table comparing 30 TB storage options shows (January 2024 prices [6, 13–15]):

Configuration	Price	Capacity		Read		Write	
	$	TB	/$	GB/s	/$	GB/s	/$
16×1.9 TB PCIe 5 SSD	6,832	30.7	4.5	176	**.03**	110	.008
8×3.8 TB PCIe 5 SSD	5,376	30.7	5.7	88	.02	52	**.010**
4×7.7 TB PCIe 5 SSD	4,620	30.7	6.6	44	.01	26	.006
8×3.8 TB PCIe 4 SSD	5,032	30.7	6.1	52	.01	28	.006
8×3.8 TB PCIe 3 SSD	3,592	30.7	**8.6**	24	.01	16	.004

Our setup (highlighted) is 6% more expensive than an equivalent setup using previous-generation PCIe 4.0 SSDs, and only 50% more expensive than a setup using PCIe 3.0 SSDs from 2019, while still providing 30× more storage capacity per dollar than DRAM. In terms of read and write bandwidth, both absolute and per dollar, PCIe 5.0 SSDs are strictly better than all older generations.
Spilling vs. distributed processing. The increase in NVMe bandwidth parallels the development of network bandwidth in the cloud: Newer AWS EC2 instances achieve up to 200 Gbit/s ≙ 25 GB/s network throughput [3]. Recent work strives to utilize these fast networks, e.g., for efficient scans from blob storage [33] or by spilling and caching data on remote memory [56, 80]. Since distributed processing closely resembles spilling – exchanging partitions parallels spilling to other nodes – we believe that the techniques proposed in this paper could be adapted to distributed query processing.

3 BACKGROUND: SPILLING ALGORITHMS

In-memory operators. Before reviewing out-of-memory query processing algorithms, let us first discuss state-of-the-art in-memory query processing algorithms using the join operator as an example. In memory, one can build a large hash table for the left join side (build side) and probe it with each tuple from the right side (probe side). In-memory systems can pipeline this probing across multiple joins and thereby avoid materializing all tuples between joins.
Running in-memory algorithms on flash storage. Given the high throughput of modern NVMe SSDs, one might consider simply using this well-known and efficient in-memory algorithm on SSDs. However, SSDs are page-based devices, where reading and writing happens at page (often 4 kB) granularity [40]. Algorithms based on point accesses, such as the hash join, thus result in read and write amplification, i.e., $\frac{\text{page size}}{\text{tuple size}}$. We can see the impact of this by building a hash table on SSD, and comparing the performance with a page-based algorithm such as partitioning. Assuming 839 M 128 Byte tuples (≙ 100 GB) and 50 GB/s I/O throughput, we get:

	Writes	Total I/O	Tuples per s	Time
Hash table on SSD	839 M	3,200 GB	6.5 M	128 s
Partition to SSD	26 M	100 GB	419.4 M	2 s

As this calculation illustrates, write amplification (here: $\frac{4\,\text{kB}}{128\,\text{B}} = 64\times$) makes random access-based algorithms impractical on SSDs. We thus need algorithms specialized for out-of-memory processing.

Disk-optimized algorithms on flash storage. The two classes of algorithms used for out-of-memory processing are hash partitioning and sorting, and some even suggest always relying on sorting [32]. Sorting is generally more useful in a disk-based setting because it facilitates a sequential I/O pattern that suits the hardware, for example by producing large runs of values to amortize disk seek time. In contrast, SSDs are inherently parallel devices, and achieving high throughput necessitates exploiting this parallelism [65].

Hash partitioning is a natural fit for SSDs. Besides sorting, one can also use hash partitioning for out-of-memory query processing. Hash partitioning-based algorithms are a better fit for the parallel nature of SSDs because they produce multiple independent partitions that can be split into fixed-size pages. Assuming T threads and P partitions, a hash partitioning operator produces $T \times P$ tuple streams. It can materialize each stream onto pages that are spilled to SSD whenever they fill up. Therefore, considering our join example, the grace hash join is a good fit for NVMe SSDs.

Asynchronous I/O and hash partitioning. Effectively exploiting the parallelism inherent to NVMe SSDs requires using asynchronous I/O interfaces [40, 78]. Our engine Spilly uses `io_uring` [17] for this. Asynchronous I/O also integrates well with algorithms based on hash partitioning. Once a page belonging to a partition is full, it can be written asynchronously while the operator continues with a freshly allocated (or written-out) page. Since hash partitioning fits SSDs well, this work focuses on hash-based operators.

4 THE UMAMI INTERFACE

Motivation: The dichotomy of real-world workloads. Only 5% of analytical queries in Snowflake's 2018 workload trace spill data, but those 5% contribute 45% of the overall CPU time and 29% of the total execution time [77]. It follows that enhancing the performance of these few spilling queries can greatly improve overall execution time. But, as is the nature of exponential distributions, the same workload trace shows that 97% of queries scan less than 5 GB. Consequently, improving spilling queries *must not* slow down this overwhelming majority of small in-memory queries.

Outline. In the following, we argue that current systems cannot effectively tackle this challenge (Section 4.1) and then describe a solution using the Umami interface. Umami leverages two independently applicable but complementary techniques called *adaptive materialization* and *self-regulating compression*, which we explain in Sections 4.2 and 4.4. We subsequently use Umami to build unified join and aggregation operators (Sections 4.5 and 4.6) that stay fast in memory but are able to switch to out-of-memory processing if necessary. As we will demonstrate in Section 6, these operators allow the query engine Spilly to execute a 10 TB TPC-H workload on a single node with little performance loss while spilling. Finally, Section 4.7 discusses the generality and limits of Umami.

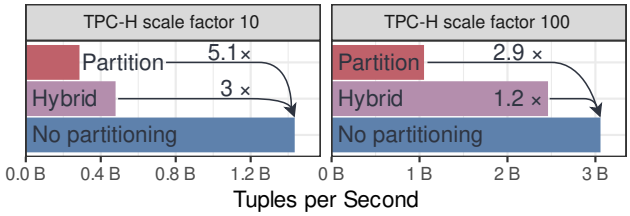

Figure 2: TPC-H performance with partitioning, hybrid, and non-partitioning operators. Data is scanned from memory.

4.1 Why the State of the Art is Not Enough

The dilemma of operator choice. State-of-the-art systems, before actually executing any query, use a query optimizer to pick physical operator implementations for each logical operator. Using an internal cost model and – often unreliable [52] – cardinality estimates, the optimizer is now faced with a choice: (1) Pick in-memory-optimized operator implementations (e.g., simple hash join, non-partitioning aggregation) that fail once memory runs out and restart the query with a different operator or plan; or (2), pick out-of-memory-capable operator implementations (e.g., grace join, hybrid hash join, partitioning aggregation) that are, at least in principle, able to spill data. If the partitioning- and sorting-based operator variants were as fast as their non-partitioning cousins, there would be no trade-off. Alas, they are not as fast.

Always partitioning is undesirable. We validate this assumption by implementing a partitioning aggregation and a grace join, as well as a non-partitioning aggregation and a simple hash join in the query engine Spilly (without sort-based algorithms, see Section 3). Figure 2 shows the throughput (scanned tuples divided by execution time) of TPC-H on scale factors 10 and 100, using the test system we describe in Section 6. The partitioning join and aggregation implementations exhibit 5× worse performance compared to the non-partitioning variants. This confirms the observation made by previous work [23, 24], which shows that, even using all modern partitioning optimizations [67, 74], simple hash joins are almost always the fastest choice. Being 5× slower in the majority of small queries, just in case one needs to spill, is unacceptable.

The hybrid hash join: A solution? There is another approach to joins, whose name suggests that it combines the benefits of both worlds: The hybrid hash join (HHJ). The HHJ was conceived in 1984 to reduce the I/O requirements of the grace join [28]. Like the grace join, the HHJ always partitions the build side of the join. Unlike the grace join, it may keep some partitions in memory to build in-memory hash tables over them [59]. On the probe side, the HHJ hashes each tuple and checks whether the partition it belongs to was spilled. If a tuple's partition was spilled, the tuple is also spilled; otherwise, it used to probe the in-memory hash table. In a third phase, the HHJ performs a grace join on the subset of partitions that were spilled. If no partitions were spilled, the third phase is omitted, and the HHJ probe phase behaves like a simple hash join.

Hybrid hash join performance. To check whether the HHJ performs as well as the simple hash join in-memory, we implement it in Spilly with all classic [28, 59] and recently proposed [44] optimizations (spill as few partitions as possible, choose spill partitions dynamically, bitmap-based probe-side check of spilled partitions), and repeat the previous experiment. As Figure 2 shows, the simple

hash join is significantly faster than the HHJ. Neither join variant spills any data in these experiments, so the probe side of both joins is algorithmically equivalent. But the HHJ still has to partition the build side, meaning more allocations, more cache misses, more TLB pressure, and more instructions.

What the hybrid hash join lacks. Fundamentally, the HHJ only provides a partial solution: It reduces I/O compared to a grace join while spilling, which is what it was originally designed for [28]. However, it does not provide in-memory execution times on par with a simple hash join. As with the grace join, being multiple times slower in the majority of small queries is an unacceptable trade-off.

The fundamental problem: Ahead-of-time operator choice. As we have demonstrated, state-of-the-art systems have to choose between being fast in-memory and allowing out-of-memory query processing. Our system evaluation (Section 6) shows that current in-memory systems choose to fail, while out-of-memory-capable systems choose to be slower. The fundamental issue is that – in general, absent special cases such as an aggregation on a single, low-cardinality column – systems *cannot know beforehand whether an operator has to spill*. If they could predict this perfectly, they could choose the correct physical operators perfectly ahead-of-time. Alas, they cannot. With *Umami*, we instead render physical operator choice unnecessary.

4.2 Adaptive Materialization

The starting point. The previous section established three things: (1) in-memory operators suffer from partitioning, but (2) out-of-memory-capable operators require partitioning, and (3) state-of-the-art systems have to decide up-front which operator implementation to use. One approach to keeping small queries fast is to restart a query with partitioning operators once an operator runs out of memory. But this is very expensive by definition: Once an operator has materialized enough data to run out of memory, it has done a lot of work that needs to be discarded. Umami avoids restarting a query and re-partitioning all data using *adaptive materialization*.

The three pillars of adaptive materialization. As the name suggests, adaptive materialization allows operators to switch their materialization strategy *during query execution*. In particular, operators can enable and disable partitioning and spilling at runtime. An operator can start without partitioning, but then enable partitioning at a later point, e.g., when memory runs low. It can then decide to materialize tuples to external storage (spill) instead of allocating new memory. All of this is dynamic, so there is no need for cardinality estimates and ahead-of-time decisions. Furthermore, previously computed results are not discarded, and compiling engines do not have to recompile code. Adaptive materialization is generic and works for all hash-based operators. This generality rests on three properties that all hash-based materializing operators exhibit:

(1) **Commonality.** All materializing (= potentially spilling) operators work in two phases: Materialize (P1) and build (P2).
(2) **Transparency.** Partitioning and spilling can be efficiently injected into phase P1 at runtime, transparent to the operator.
(3) **Independence.** Phase P2 can always apply an algorithm that is unaware of partitions, even if phase P1 partitioned.

To understand why and when Umami's adaptive materialization works, we explain these properties in the following.

Listing 1: Injecting partitioning logic into operators. Changes required for Umami are highlighted.

```
//  Umami's operator-independent materialization  //
u64 shift = 64
void storeTuple(void* tuple, u64 size , u64 hash )
  Page& page = this->output[hash >> shift]
  if (page.isFull(size)) { b.getEmptyPage(page) }
  memcpy((page.cursor += size), tuple, size)
```

```
//   query-specific operator code (generated)   //
UmamiBuffer& b = operatorStorage.threadLocal()
for (auto* tuple : input) // phase (1): materialize
  // ... <- operator-specific generated code
  b.storeTuple(tuple, tuple.size(), hashKey(tuple))
operatorStorage.finalize() // phase (2): build
for (auto& page : operatorStorage.tuplePages) // ...
```

Commonality. Let us begin by examining the common high-level structure of in-memory operator algorithms implemented by state-of-the-art systems [23, 51]: Initially, operators materialize all input tuples into thread-local memory regions, optionally filtering or pre-aggregating them. Due to the unpredictability of the input size, these memory regions are organized as pages and allocated as needed. Consequently, after materializing all data, the operator holds a list of pages storing tuples. In a subsequent phase, operators construct a data structure over these tuples: A hash map with duplicates for joins, a regular hash map for aggregations, a segment tree for window functions [54], etc. Therefore, every materializing operator can be thought of as operating in two distinct phases: A *materialization* phase and a data structure *build* phase.

Transparency. The materialization phase exhibits a small yet pivotal detail that enables the injection of partitioning and spilling logic at runtime: As tuples are materialized into dynamically allocated memory, operators cannot presuppose any specific memory location for the tuples, i.e., tuple locations are transparent to the operator. For instance, if the (n)th tuple fills up a page, then the $(n+1)$th tuple will be placed not adjacent to it, but on an entirely different page. Listing 1 illustrates this: Umami's materialization logic (`storeTuple`) checks whether a page has space, allocates a new page if necessary, and finally copies the tuple to the page.

Injecting partitioning logic at runtime. Since tuple locations are transparent to the operator anyway, the materialization logic can also place the tuple in a different location based on the tuple hash, i.e., it can apply hash partitioning. During materialization, the operator need not know whether hash partitioning is applied, as long as it adheres to the *transparency* property. Thereby, hash partitioning is abstracted from the operator. Based on this, *adaptive materialization* in Umami can enable and disable partitioning at runtime, without separate branches for each case: By keeping page pointers in a contiguous memory region ("page array"), Umami can choose whether to partition and how many partitions to use by adjusting a `shift` value that is applied to the hash and used to access the page array, which it allocates on demand. Listing 1 illustrates this: If the shift value is 64, the hash shifts to zero, and there is no partitioning. If the shift is less than 64, the hash shifts to a larger index in the page array, resulting in $2^{64-\text{shift}}$ partitions.

Listing 2: Spilling as allocation (top), Umami buffer (below).

```
void SpillBuffer::getEmptyPage(Page& page)
  auto diskLoc = io.queueForAsyncWrite(page.data)
  spilledPageLocations[page.part].push_back(diskLoc)
  if (pool.empty()) { io.pollDoneWrites(&pool) }
  page.data = pagePool.popPage() // get clean page
  page.cursor = page.data // reset page write cursor
```

```
struct UmamiBuffer { // ↼ per thread and operator
  Page* output // ↼, ↓ fast page access interface
  void storeTuple(void* tuple, u64 size, u64 hash)
  virtual void getEmptyPage(Page& p, u64 part)
}; // ↑ slower polymorphism only during allocation
struct PartitionBuffer : UmamiBuffer {...} // ↼ ext-
struct SpillBuffer : UmamiBuffer {...} // ↼ ensible
```

Independence. After materializing all data in phase P1, an operator using Umami's adaptive materialization interface finds itself in one of two states: It either (1) did not partition anything and holds only pages with unpartitioned data or (2) holds a mixture of pages, some with partitioned data and others with unpartitioned data. At first glance, one would assume that an operator in state (2) has to re-partition the unpartitioned data before continuing. If, however, all data still resides in memory, the operator can employ the same algorithm as it would in state (1) and avoid re-partitioning. For instance, a join can – rather than partially re-partitioning and executing a grace join – simply allocate one large hash table and execute a simple hash join over the mixed partitioned and unpartitioned data.
Deciding whether to partition. At runtime, Umami needs to decide whether to start partitioning. This decision can rely on metrics such as the available memory, the unique value count of seen tuples, or the percentage of data yet to be materialized. Umami co-locates this decision-making logic with page allocation, i.e., `getEmptyPage` in Listing 1. This amortizes the runtime cost of the adaptivity logic over the number of tuples in each page. To decide on partitioning, Umami's implementation in our engine Spilly uses a heuristic based on the allocated memory size, which we explain in Section 5.3.
Deciding whether to spill. Deciding on whether to partition during page allocation has an additional benefit: Allocation is also the perfect place to decide on whether to start spilling. When allocating pages (`getEmptyPage`), threads check whether the memory budget is exhausted, and, if so, they spill the full page to SSD instead of allocating a new page. Because writes are asynchronous, the spilled page cannot be reused until the write finishes. Instead, Umami has to either allocate a new page or check for finished writes of previously spilled pages. As Listing 2 (top) illustrates, Umami handles both cases efficiently using a thread-local pool of pages which operators can draw from. This bounds memory usage and ensures that I/O occurs in the background while query processing can continue.

4.3 Umami Generalizes the Hybrid Hash Join

Umami's interface design. Umami's adaptive materialization causes little overhead at runtime (c.f., Section 6) while also being flexible enough to allow for different allocation, partitioning, and spilling strategies. Umami achieves this by keeping the interface that is invoked for each tuple during query execution fast and carefully placing extension points on code paths that are not performance-critical. The fast interface is a data structure with efficient random access for switching between pages. In C++, this is simply the `output` array in Listings 1 and 2 (bottom). On non-critical code paths, such as allocation, Umami stays flexible using runtime polymorphism, e.g., `getEmptyPage` in Listings 1 and 2. This means that Umami's adaptive materialization interface is extensible by simply adding new classes that implement the same interface.
From hybrid hash join to generalized hybrid spilling. The adaptive materialization interface can generalize the hybrid hash join (HHJ) spilling approach to other operators. The core idea of the modern hybrid hash join is to lazily pick a subset of partitions to spill based on heuristics (e.g., choose the largest partition) so that more partitions remain entirely in memory. The join's probe side then queries which partitions were spilled using a bitmask [44, 59]. Umami can easily mimic this by adding a new `UmamiBuffer` variant (c.f., Listing 2, bottom). This variant picks partitions to spill in `getEmptyPage` using the HHJ heuristics and exposes a bitmask after materialization, which works for *any* operator that uses Umami. Spilling occurs at page granularity, which suits page-based devices like NVMe SSDs [40]. Spilly's operators (Sections 4.5 and 4.6) automatically benefit from Umami's generalized hybrid spilling approach. Section 5.3 discusses further implementation details.
Tuple-based heuristics. Fine-grained, per-tuple decisions about what to keep in memory and what to spill can further reduce I/O. Histojoin and NOCAP [16, 81], for instance, use correlation data to keep those build-side tuples with the most probe-side matches in memory. This can be done independently of Umami, e.g., by checking a hash set for each key before materialization, which can impact in-memory performance but reduce I/O in skewed workloads.

4.4 Self-Regulating Compression

Compression in Umami. The previous sections describe Umami's adaptive materialization interface, which enables dynamic partitioning and spilling at runtime. This improves upon previous approaches by enabling fast in-memory operators to spill to SSD if necessary. But state-of-the-art systems also have to handle evolving and varying hardware that challenges old assumptions, e.g., that I/O is slow and CPU is never the bottleneck. For example, servers may have anywhere from one to sixteen NVMe SSDs, producing an order of magnitude throughput discrepancy. We propose a solution to this challenge, which we name *self-regulating compression*.
Fast I/O reduces the benefits of compression. Compression has classically been used to reduce the size of data on disk, or to increase the effective I/O throughput. Reducing on-disk data size matters little to us because spilled data is ephemeral. Increasing the effective I/O throughput is an interesting prospect, but has to be considered carefully: With high-throughput SSDs, I/O is not always the bottleneck, and the CPU cost of compression may even deteriorate performance. The effectiveness of compression for spilling thus depends on the storage throughput available to the engine.
A common cost metric for CPU and I/O. The effectiveness of compression also varies from operator to operator: If an operator is I/O-bound, using the spare CPU time for compression benefits performance; if it is compute-bound, however, compression just adds to the CPU cost and deteriorates performance. In other words,

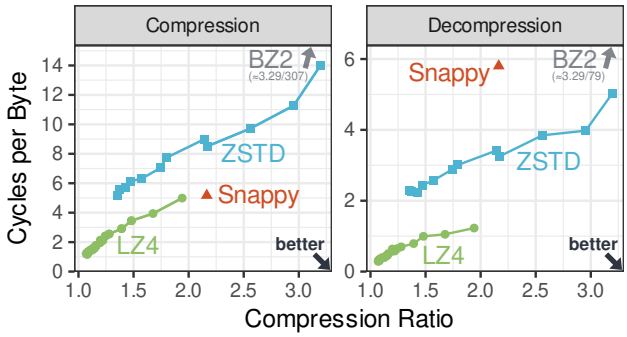

Figure 3: Compression ratio vs. (de)compression cost on spilled TPC-H tuple data using Snappy, BZ2, and multiple settings for LZ4 and ZSTD.

compression allows I/O-bound queries to trade spare CPU time for I/O time. But are queries ever I/O-bound when using modern SSDs? To answer this question, we introduce a common metric that measures the cost of computations, compression, and I/O: Cycles per Byte. Determining this metric for queries, compression, and I/O will allow us to build a cost model for trading CPU and I/O time using compression. First, we measure the CPU cycles per scanned byte in our query engine Spilly across all TPC-H queries. We find that cycles spent per byte vary by up to 20× between queries:

TPC-H SF 100	Q1	Q13	Q16	Q17	Q19	max➔min
cycles/byte	3.3	**60.3**	37.2	**3.0**	4.3	20.2×

The cost of I/O. One can compare the CPU cost of queries with the cost of I/O by calculating how many cycles it takes to spill one byte. On our test system (c.f., Section 6.1), the cost for writing one byte equals $\frac{96\text{ cores}\cdot 3.5\text{ GHz}}{8\text{ SSDs}\cdot 6\text{ GB/s}} = 7.38$ cycles. When adding the cost of reading back the spilled data, the combined I/O cost per byte is 11.1 cycles. This cost is within range of the TPC-H query costs we determined, meaning some queries will be I/O- and others compute-bound *even when spilling*.

Evaluating compression effectiveness in TPC-H. We have determined that compressing spilled data can sometimes be useful in principle, but it is not yet clear whether compression is cheap enough in practice. In addition to CPU and I/O cost, we thus measure the cost of the general-purpose compression schemes LZ4, Snappy, ZSTD and BZ2 using their open-source libraries. Each scheme is applied to the spilled pages produced by Umami across all 22 TPC-H queries. Figure 3 shows the measured average compression ratio, as well as compression cost (left) and decompression cost (right) as cycles per byte. Because we test multiple compression settings offered by LZ4 and ZSTD, these schemes are represented by multiple points in the figure. We find that (1) the *compression cost* of all schemes, except BZ2, is similar to CPU and I/O costs, making these schemes affordable, (2) the *compression ratio* is significant, even on synthetic TPC-H data, and (3) the compression cost/ratio *trade-off curve* is smooth with few outliers, which suits optimization algorithms. Based on these insights, we design a compression algorithm that regulates itself at runtime to increase performance.

Matching CPU and I/O bandwidth. A self-regulating compression algorithm has two primary tasks: First, detect when it makes

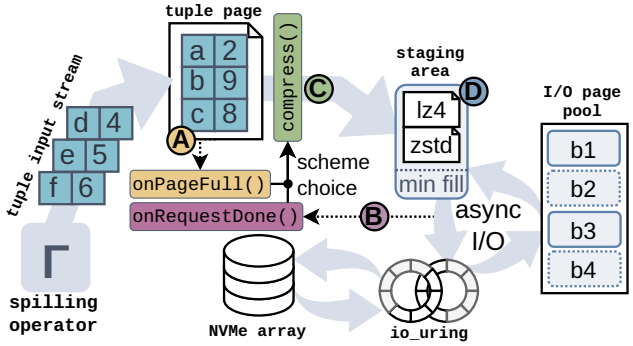

Figure 4: Compression and I/O in a spilling operator.

sense to compress. Second, decide on the compression scheme and settings, which determine the trade-off between compression cost and compression factor, as Figure 3 shows. One can solve both tasks at once by modeling a *bandwidth equilibrium* that needs to be reached. A regulating algorithm continuously measures operator, compression and I/O bandwidth; if the combined CPU bandwidth (operator plus compression) and I/O bandwidth are not equal, the algorithm changes the compression setting until the effective (compressed) I/O bandwidth matches the CPU bandwidth.

Tracking bandwidths. Adjusting compression based on I/O and CPU bandwidths requires lightweight tracking mechanisms for I/O and CPU throughput. However, we found tracking the inverse, *cost* (cycles per byte), to be more natural than tracking bandwidth (bytes per cycle): Tracking CPU cost is as trivial as timing how long it takes to process each page (Ⓐ in Figure 4). Tracking I/O cost is more difficult because multiple I/O requests happen asynchronously and in parallel on each thread. One can do this efficiently by encoding the I/O request start time in each `io_uring` user data field [17] and calculating the average cycles per byte using the measured latency and number of simultaneous I/O requests (Ⓑ in Figure 4).

Self-regulating compression algorithm. Let us now walk through the self-regulating compression algorithm, outlined in Listing 3 (Ⓒ). Given an initial scheme choice, i.e., *'Uncompressed'* or *'LZ4 with parameter 16'*, the algorithm compresses incoming pages and measures the compression ratio and CPU cycles. After a *run* of N pages (default: $N = 2 \cdot \text{iodepth}$), the algorithm compares the average CPU (operator plus compression) cost of that run with the average I/O cost (cycles per *compressed* byte). If I/O cost outweighed CPU cost, compression is increased, otherwise reduced. The algorithm then continues with the new compression setting for the next N pages.

A unified scale. Increasing and decreasing compression may involve switching to a different scheme based on the experimentally determined trade-off curve shown in Figure 3. This experiment rules out Snappy (outside pareto-frontier), BZ2 (too expensive), and the settings of ZSTD for which LZ4 is strictly better. The remaining LZ4 and ZSTD settings are mapped to a unified scale (c.f. Listing 3, top), allowing for simple and efficient scheme switching.

Discussion: Why general-purpose schemes? Like most query engines, Spilly uses a columnar storage format for table data (see Section 5.2) and a row-wise format for materialized tuple data. The reason is that operators can consume row-wise tuples directly, e.g., by linking to them from a hash table [51], and access them with better locality. Since using Umami should not deteriorate in-memory

Listing 3: Self-regulating compression and its integration.

```
enum Scheme : u8 { Uncompressed=0, LZ4=1, ZSTD=2 };
union Scale { u16 optVariable; // <- optimize
/*output ->*/ struct { u8 param; Scheme scheme; }};
```

```
class SelfRegulatingCompression: // per thread
u64 pageTimer; Scale curScheme; CompressionRun run
ExpAvg pipelineAvg, latencyAvg; // track costs
void onPageFull(u64 size) Ⓐ // pipeline cost
  pipelineAvg.add((now() - pageTimer) / size)
  pageTimer = now() // reset for next page
void onRequestDone(u64 uringReqId) Ⓑ // IO cost
  latencyAvg.add(now() - decodeTime(uringReqId))
size_t compress(Page& in, Page& out) Ⓒ
  u64 start = now() // track compression cost
  size_t comprSize = curScheme.compress(in, out)
  run.track(in.size, comprSize, now() - start)
  if (++run.count == iodepth) { selectNextScheme() }
  return comprSize
void selectNextScheme() // adaptive scheme selection
  f32 ioCost = latencyAvg / iodepth / run.comprRatio
  f32 cpuCost = run.comprCost + pipelineAvg
  int diff = ioCost - cpuCost // cpu time left?
  curScheme.optVariable += clampMinMax(diff)
  startNewRun() // test next run with new scheme
```

performance, it is important to allow operators to keep using their own optimized row-wise tuple storage format when materializing data. This necessitates using general-purpose compression schemes instead of columnar compression schemes. Another advantage of using general-purpose schemes is that Umami stays generic and operator-independent. Since Umami uses standard, general-purpose compression libraries, it will also benefit from hardware accelerations targeting those compression libraries [10, 11]. Still, testing layout-aware row-wise compression schemes [79] and other page layouts such as PAX [19] could improve compression ratio and performance, and is an interesting avenue for future work.

Reacting to outside influences. The self-regulating compression algorithm measures and adapts to all parameters – compression cost, operator CPU time, I/O latency – at runtime. Its only predetermined parameter is the compression scheme scale, i.e., Uncompressed > LZ4 > ZSTD. The resulting flexibility enables it to automatically react to outside influences as well. For example, when running Umami on a rented cloud VM, SSD performance may be influenced by "noisy neighbors", i.e., SSD usage from other VMs running on the same physical machine. Since the I/O latency is measured at runtime, self-regulating compression should be able to automatically optimize the compression in this scenario as well.

4.5 Unified Hash Join

Applying Umami. Let us now discuss how Umami can convert a state-of-the-art in-memory join operator to a spilling join that effectively utilizes NVMe arrays. We call it *unified hash join* because it subsumes all hash-based join variants we know, and because it unifies in-memory and out-of-memory processing. Spilly uses it to implement inner, semi, anti, and outer equi-joins.

Baseline: The simple in-memory hash join. Our starting point is an in-memory hash join implemented by state-of-the-art systems [23, 24, 51]. It uses morsel-driven parallelism to concurrently materialize tuples into thread-local buffers (phase P1), and later builds a multi map implemented by a chaining hash table (phase P2). Since the bucket count in a chaining hash table is proportional to the unique value count (not the absolute count), the hash join uses another optimization: It builds a HyperLogLog sketch [34] during materialization to estimate the unique value count, producing better hash table sizes for joins with duplicates on the build side.

Phase 1: Materialization. Using the Umami interface, it is simple to enable out-of-memory processing during the join materialization phase. For each input tuple, the join "calls" `storeTuple` (c.f, Listing 1), which is inlined into the generated code. The HyperLogLog sketch already computes a hash, which is passed to `storeTuple` and thus reused for dynamic partitioning. The page allocation logic dynamically switches to spilling and partitioning, unbeknownst to the operator. To summarize, applying Umami changes almost nothing in the operator code during the materialization phase.

Phase 2: Building and probing a hash table. At the start of the second phase, Umami might have (A) materialized without partitioning, (B) partitioned the data or (C) partitioned and spilled. In cases (A) and (B), the join operator can continue with a simple hash join: It builds a single large hash table over the materialized data, automatically benefitting from improved locality if the data is partially partitioned as in case (B) (see Section 5.3). If data was spilled (case C), the operator executes its second phase like a hybrid hash join, with hybrid spilling implemented by the Umami framework. The join probe side then uses Umami to partially partition tuples like a hybrid hash join, automatically spilling pages if necessary.

Overview: Unified Hash Join vs. hybrid hash join. Let us briefly review how the hash, grace, hybrid, and unified join variants differ in terms of partitioning and materialization logic. The hash join (HJ) only materializes the build side and streams the probe side; it partitions nothing. The grace join (GJ) materializes and partitions both sides and executes a hash join for each pair of partitions. The hybrid hash join (HHJ) always partitions the build side, and partitions only probe-side tuples whose partitions were spilled on the build side. The unified join (UHJ) adaptively partitions and spills the build side using Umami and executes the probe side like a HHJ.

4.6 Unified Hash Aggregation

Designing a unified aggregation operator. We now discuss an aggregation operator capable of high-performance out-of-memory query processing on NVMe arrays. As with the join, it is crucial that the in-memory variant of the operator be fast as well.

Adapting the aggregation to output cardinalities. Following the reasoning from Section 4.1, an in-memory aggregation operator should be hash-based and not use partitioning for small inputs. But unlike the join operator, the aggregation operator can benefit from using different in-memory aggregation algorithms based on the unique value count (cardinality): (1) for small cardinalities, threads should locally preaggregate data to avoid contention and keep data in the CPU caches; (2) for medium cardinalities, local preaggregation is less effective – larger thread-local tables or a single global hash table are preferable; (3) for large cardinalities, partitioning improves cache locality; (4) for larger-than-memory cardinalities,

partitioning is necessary; preaggregation can be useful for reducing data size before writing to SSD. These trade-offs have been studied previously for in-memory processing [27, 51]. Taking them into account, we build a robust in-memory aggregation operator that scales seamlessly to out-of-memory query processing using Umami.

Small cardinalities: Local preaggregation. To efficiently aggregate inputs with few unique values (e.g., TPC-H Q1), small, thread-local hash tables sized to fit into the L1-cache preaggregate data. Like the join, the aggregation operator uses per-thread HyperLogLog sketches to estimate the cardinality. After consuming the input, it combines the sketches to determine the size of a global synchronized hash table into which threads merge preaggregated tuples. This merging phase uses the morsel-driven scheduler, which allows work-stealing to accommodate any potential data skew.

Large cardinalities: Partitioning. For larger cardinalities, the cache-resident thread-local hash tables will eventually fill up. Once this happens, the preaggregated tuples have to be evicted either onto partition pages or into a larger hash table. Estimating the correct hash table size from an in-progress HyperLogLog sketch without having seen all tuples is error-prone and difficult, and rehashing due to wrong estimates can be expensive. The aggregation operator thus evicts the preaggregated tuples onto partition pages instead. To avoid copying tuples on eviction, thread-local hash tables write tuples into these pages directly and only store an offset into each page. "Eviction" of tuples is done by simply invalidating the respective thread-local hash bucket range once a page is full.

Applying Umami. At this point, the straightforward strategy of storing the partially preaggregated tuples on pages using adaptive materialization proves effective: After consuming the input, the operator reads tuples from Umami-managed pages and aggregates them to a global synchronized hash table as before. With this, the aggregation operator is capable of NVMe-optimized out-of-memory query processing. Umami transparently switches to partitioning and spilling if necessary, re-partitions if necessary, and reads pages from NVMe array if necessary. Based on HyperLogLog sketch cardinality estimates, the aggregation operator can either aggregate tuples to a single large hash table or aggregate partitions independently. Building the hash table also automatically benefits from the locality and contention optimizations described in Section 5.3.

Comparison with DuckDB. Interestingly, Spilly's aggregation operator closely resembles DuckDB's recently published external aggregation operator [48]. Both accelerate small-cardinality queries using thread-local hash tables with linear probing whose hash buckets point to pre-aggregated tuples stored on pages for spilling. DuckDB's operator differs mainly in that (1) thanks to Umami, Spilly can start partitioning lazily only if necessary, (2) DuckDB uses the buffer manager for intermediary results, whereas Umami's buffers are managed independently and per-operator, and (3) due to that, Umami can evict pages and evice thread-local hash table bucket ranges to partitions at will, whereas DuckDB unpins pages referenced by the hash table in the buffer manager. Earlier work [42] has proposed more sophisticated group eviction mechanisms.

4.7 How General is Umami?

Non-hashing Operators. Section 4.2 introduces adaptive materialization, a technique which only requires a hash key to dynamically and transparently enable partitioning and spilling at runtime. Hash joins, hash aggregations, hash-based window functions [54], and any other hash-based operators can all take advantage of adaptive materialization. Applying adaptive materialization to other operators, such as sorting, is an interesting direction for future work, especially considering the similarities between sorting and hashing [32, 58]. Umami's self-regulating compression (Section 4.4) and efficient spilling are not hash-specific and work with any operator.

Transactional workloads. HTAP systems need to balance transaction latency with analytical throughput. This extends to out-of-memory query processing, since the write-ahead log, page eviction, and spilling of intermediary results all compete for the available SSD bandwidth. Besides changing the number of threads designated to OLAP vs. OLTP, one can conceive of several other ways to allot SSD bandwidth. For example, one could designate some SSDs for transactional processing, and the rest for spilling. It is also possible to cap Umami's I/O throughput by limiting the maximum allowed I/O parallelism. Lastly, one could configure self-regulating compression to compress more aggressively to reduce I/O. Evaluating these techniques is an interesting avenue for future work.

5 IMPLEMENTATION

Implementation matters. Using the Umami interface allows our query engine Spilly to seamlessly spill data to modern NVMe SSDs without significantly complicating operator implementations. However, achieving high performance also requires careful engineering. In the following, we discuss some important implementation details that enable high-performance query processing in Spilly.

5.1 Utilizing NVMe Arrays

What a modern I/O stack should provide. Spilly features an NVMe-optimized, asynchronous I/O implementation that effectively utilizes the modern I/O interfaces of the OS, effectively utilizes multiple NVMe SSDs, integrates with self-regulating compression (c.f., Section 4.4). We describe its design in the following.

Utilizing `io_uring`. An I/O stack that provides the interface assumed in Listing 2 can be implemented as a thin wrapper around `io_uring`. Each thread manages its own ring to avoid contention. Since individual writes are not latency-critical, threads collect write requests in their local submission queue and flush them to the OS as a batch. Threads track the number of outstanding requests to ensure correctness and bound memory usage. Once an asynchronous I/O request is done, the associated page is returned to the operator.

Supporting multiple SSDs. To spread reads and writes across multiple SSDs, each thread tracks which SSD it wrote to last and assigns the next write in a round-robin fashion. Thus, write spreading is fully decentralized and the only coordination point is a single per-SSD atomic counter which tracks the end of the written-to section in the designated on-disk spilling area. This counter is necessary to make sure writes from different threads do not overlap.

5.2 NVMe-Optimized Scan Operator

Scan features. The asynchronous I/O stack described in Section 5.1 can fully and efficiently utilize multiple PCIe 5.0 NVMe SSDs. Spilly's scan implementation is based on the same I/O stack, is compatible with the row-group data layout prevalent in modern OLAP systems, and integrates well with morsel-driven parallelism.

In-memory vs. external scans. Parallel scans in in-memory engines are trivial: Worker threads fetch [index, length] pairs which they use to access the in-memory arrays containing the columns. After consuming a chunk of tuples ("morsel"), a thread can immediately fetch the next morsel and start processing by dereferencing a pointer. External scans from an NVMe array, on the other hand, are more complex: Before processing a single tuple, each thread has to asynchronously read column chunks from multiple different SSDs to achieve maximum throughput. It then has to reconstruct rows across chunk boundaries, while simultaneously starting requests for the chunks containing the tuples it needs to process next.

Data layout optimized for NVMe arrays. Maximizing scan throughput even for a single column requires distributing columns across SSDs and storing them in multi-kilobyte chunks. This approach results in on-disk chunks that contain differing numbers of values, depending on the size of the values. But for processing, each thread has to efficiently reconstruct complete tuples from these column chunks while reading. This is usually done by dividing the data into row groups, which we size at 32k tuples by defaultRow groups simultaneously serve as the unit of parallelism (morsel [51]): Each thread fetches a morsel consisting of one or more row groups, schedules read requests for the pages within the row groups, aiming to maintain a full I/O queue, and processes fully read row groups.

Weakening morsel boundaries to improve I/O throughput. While this row groups-as-morsels approach simplifies storage and reconstruction of tuples, it has a major downside: Before fetching the next morsel, a thread must wait for all I/O requests to finish and finish processing the last row group in the current morsel. Consequently, the asynchronous I/O queue is completely drained at each morsel boundary, reducing I/O parallelism and, therefore, throughput. To improve upon this, threads should start I/O requests for the next morsel(s) while still processing the current morsel. However, this is not possible in the classical morsel-driven paradigm (e.g., abstractions like `parallel_for`). For this reason, we implement an enhanced morsel-based scheduler that weakens morsel boundaries, allowing threads to optionally "prefetch" morsels and start I/O requests for them while still processing previous morsels.

Table compression. Table compression can speed up scan-bound queries by a large factor, i.e., the compression ratio. We apply the recently proposed columnar compression algorithm BtrBlocks [49] as an off-the-shelf solution. The resulting compression ratio is similar to that of other state-of-the-art systems:

	Col. Store S	DuckDB	Spilly with BtrBlocks
SF 10k Size (TB)	3.36	2.65	3.39
Compression (×)	2.97	3.77	2.95

5.3 Adaptive Materialization Implementation

Exploiting locality without partitioning. Partitioning has the often-cited benefit of improving locality [22, 67, 73]. For example, building a small hash table over a subset of the data results in more cache-friendly memory accesses compared to building a large hash table over the entire data. This raises the question of whether avoiding re-partitioning, as described in Section 4.2, could sometimes be worse than the alternative. It turns out that one can get the best of both worlds: We allow operators to provide *locality hints* to the scheduler, meaning each thread prefers processing a particular range of pages. The scheduler only obeys these hints opportunistically to avoid impairing morsel-driven parallelism [51] and work stealing. Using locality hints, operators can ensure that each thread mainly processes pages from one partition at a time. This is particularly useful during hash table construction, for example: Using a hash based on powers of two, such that the partition bits B_P are a prefix of the hash value bits B_H, tuples from each partition are confined to a small ($2^{|B_H|-|B_P|}$) range of hash buckets. Thus, when a thread reads a partition page to insert its tuples into the hash table, it will experience better cache locality. Moreover, since different threads generally process different partitions, this method also reduces concurrency-induced hash bucket contention.

Partition sizes. In order to decide on whether to start partitioning, Spilly uses a heuristic based on our performance analysis of small queries, which indicates that most of the initial overhead of partitioning stems from memory allocation: Given an initial page size S_0, set to increase exponentially with each allocation, i.e., $S_1 = \alpha \cdot S_0$, and a partition count setting P: If $\alpha \cdot S_N \geq P \cdot S_0$, Spilly switches to partitioning, allocating P pages of size S_0 instead of one page with size $S_{N+1} = \alpha \cdot S_N$. With this heuristic, most small queries will not partition at all in Spilly. Thus, the partition count P can be set to a large number that would usually deteriorate small query performance, but improves locality and memory usage for larger spilling queries. Spilly uses 256 partitions by default, which we have experimentally determined to offer a good trade-off.

Hybrid spilling implementation details. In the following we describe some important details of hybrid spilling approach employed by Spilly, which is based on the dynamic HHJ [44, 59]. Firstly, favoring some partitions for spilling over others requires cross-thread synchronization. We do this using a bitmask protected by an optimistic lock [55], which tracks spilled and in-memory partitions. Each thread chooses an in-memory partition to spill when it reaches a configurable memory budget. Threads prefer spilling partitions that other threads have also already chosen for spilling, otherwise they choose the largest partition as suggested by previous work [36, 44]. If a thread chooses a new partition to spill, it updates the synchronized bitmask, scrapping its partition choice if the bitmask has been updated by another thread in the meantime (optimistic concurrency). The join probe side first builds a single hash table over in-memory pages across partitions in a first phase and then joins spilled pages partition-wise in a second phase.

Data format. Spilly stores fixed size tuples on pages consecutively like an array, and variable-size tuples using a slotted page layout. As a research prototype, Spilly does not support objects larger than the internal page size (64 KiB). When spilling, some additional metadata is required: First, the on-disk location of each page is stored as a [deviceId, offset, size] tuple in a single 64 bit integer. Page offset and size can be encoded compactly because they must be multiples of the device block size (512 bit) if the SSD is attached directly. Second, multiple pages may be compressed into a single staging area (c.f., Figure 4) that is written out when it contains 64 KiB of data or more. Staging areas also use a slotted page layout, where each slot stores the page offset, size, and the self-regulating compression scheme type with which the page was compressed (c.f., Section 4.4).

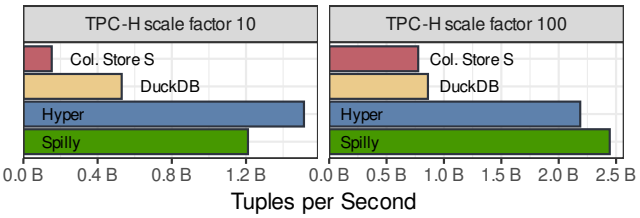

Figure 5: TPC-H SF 10–100 in-memory hot run performance.

6 EVALUATION

Outline. After explaining the experimental setup, we evaluate the end-to-end system performance on TPC-H. We then evaluate the performance of single operators and components in detail.

Spilly and state-of-the-art systems. The query engine Spilly compiles queries to C++, implements the unified operators using Umami, and is capable of running all 22 TPC-H queries. We pick multiple state-of-the-art systems to compare against to cover a wide range of the current query processing technology landscape. First, an industry-leading proprietary column store (referred to as Column Store S). Second, the columnar analytics database DuckDB [69], which recently introduced partial out-of-memory support [18, 48]. Third, the high-performance in-memory database Hyper, using its publicly available binary [5]. All three comparison systems can scan data from disk but also implement an in-memory buffer cache. Only Column Store S and DuckDB support spilling intermediary results to disk. For our purposes, Hyper has the same features as Umbra (very fast in memory, no spilling to disk), but Hyper is publicly available and therefore preferable for comparisons.

6.1 Experimental Setup

Hardware. We perform all experiments on an AMD EPYC 9645P machine with 96 cores (192 hardware threads) running Linux 6.5.0-14. It has 384 GB of DRAM capacity and 128 PCIe 5.0 lanes, which drive 8 × Kioxia CM7-R SSDs, each with 3.84 TB storage capacity. In microbenchmarks, each SSD achieves 11 GB/s read and 6.2 GB/s write throughput using 64 KiB pages.

System and measurement setup. As per their benchmarking guidelines [2], we use the latest available Hyper (0.0.18441) and DuckDB (v0.9.3-dev2533) builds. This DuckDB version already uses its recently published external aggregation operator [48]. Since we measure execution times without compilation times, Hyper is set to always compile code at the highest optimization level. *Cold runs* are executed by restarting the engine and clearing the OS page cache before each query, as suggested by recent work [68]. *Hot runs* execute the same query immediately after a cold run.

Performance metrics. To visualize and relate performance across a large range of data sizes, we report most performance numbers as "tuples per second". This number is computed per query by dividing the number of scanned tuples by the execution time.

Storage configuration. All systems store and spill data on a RAID-0 array with the XFS file system, which Haas et al. [39] found to be fastest. Unlike Hyper and DuckDB, Column Store S can manage multiple storage devices, but a RAID-0 array was faster on our setup. Spilly can attach block devices directly or use a filesystem, but the performance difference is small.

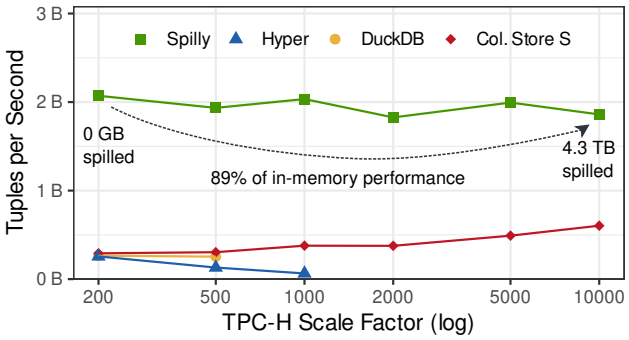

Figure 6: TPC-H SF 200 – 10,000 cold run performance.

Impact of query plans. To separate the performance impact of the logical operator ordering and the physical operators themselves, Spilly uses the operator ordering of Column Store S. Spilly's *physical* operators use Umami, so they can adapt during query execution.

6.2 TPC-H Performance

In-memory performance. We first verify that Spilly's in-memory performance is comparable to that of state-of-the-art in-memory systems. Since other systems keep all data in memory if possible, we also extend Spilly's scan operator with a simple buffer cache using a random eviction policy. We then execute "hot runs" for each TPC-H query as described in Section 6.1. Because hot runs execute a query twice, taking only the second measurement, each tested system can cache most table data in memory in this experiment. As Figure 5 shows, Spilly achieves similar in-memory tuple throughput as Hyper. We thus achieved our goal of having excellent in-memory performance while making the engine out-of-memory-capable.

Out-of-memory performance. For analytics on uncached or large data, all data must be loaded from the NVMe array. We evaluate the performance of this scenario by executing "cold runs" for each TPC-H query as described in Section 6.1. Cold runs prevent systems from caching data in memory, isolating query engine performance from buffer cache performance. All systems store data using the XFS filesystem on a RAID-0 array. Figure 6 shows the cold run performance for TPC-H scale factor (SF) 200 to 10,000. Spilly begins to spill data on SF 500, starting with 48 GB (≙ 6.5% of scanned data) and increasing to 4.3 TB on SF 10,000 (≙ 29% of scanned data):

SF	200	500	1,000	2,000	5,000	10,000
Spilling queries	0	1.0	1	2	5	10
Spilled GB	0	48	240	538	1,820	4,283
Scanned GB	274	732	1,467	2,942	7,424	14,994
Spilled fraction	0%	6.5%	16%	18%	25%	29%

Surprisingly, the overall performance drop over this wide range of data and spill sizes is only 11% in Spilly.

Performance of other systems. Hyper processes more than 1 billion tuples per second in memory, but drops to the performance of DuckDB and Column Store S when scanning from SSD. Neither DuckDB nor Hyper scale to SF 10,000. Only Column Store S is able to scale to TPC-H SF 10,000 , but is comparatively slow even when it does not spill. This is in part because it cannot consistently achieve the I/O throughput of a single PCIe 4.0 SSD, even when reading

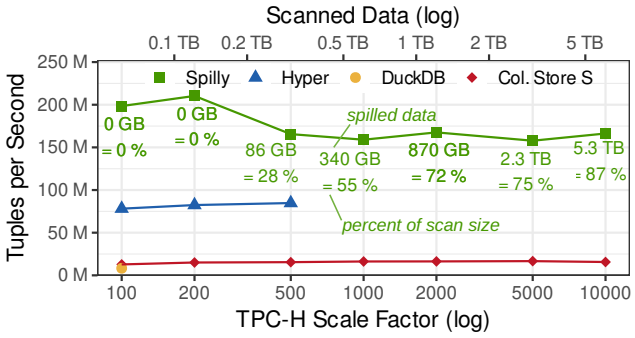

Figure 7: Aggregation microbenchmark tuple throughput and Spilly spill size. Data is scanned from SSD.

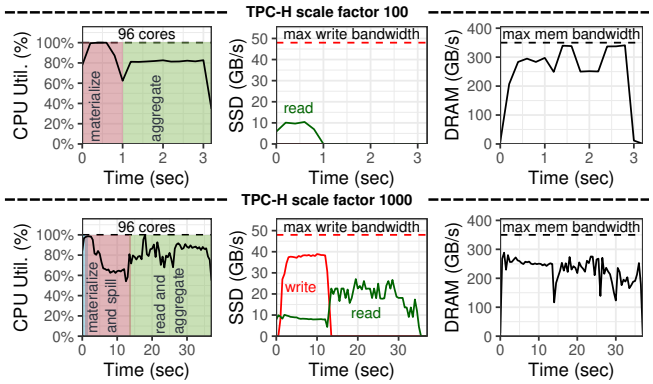

Figure 8: Aggregation microbenchmark execution trace on TPC-H SF 100 (top) and SF 1000 (bottom). Data resides on SSD.

from eight PCIe 5.0 SSDs. Interestingly, Column Store S achieves a higher average throughput on higher scale factors because the scan performance improves: The average SSD read bandwidth is 1.1 GB/s on SF 100, but 5.1 GB/s on SF 10,000. The average write I/O throughput was even lower (1.1 GB/s on SF 10,000), and we observed sort-based operators in Column Store S query plans for large scale factors, indicating an HDD-optimized approach.

Absolute times. Figure 6 visualizes system scalability across scale factors, but does not provide individual query times. The following table shows Spilly cold run times for SF 100, 1000, and 10,000:

	Response Time (s)				Response Time (s)		
Query	SF 100	SF 1k	SF 10k	**Query**	SF 100	SF 1k	SF 10k
1	0.306	2.03	19.6	**12**	0.298	1.88	20.7
2	0.208	0.584	3.19	**13**	0.660	4.84	59.0
3	0.540	2.96	46.7	**14**	0.211	1.05	9.69
4	0.270	1.67	16.4	**15**	0.239	0.966	9.50
5	0.465	3.22	40.0	**16**	0.354	1.40	44.4
6	0.143	0.621	5.45	**17**	0.287	1.76	26.5
7	0.334	2.37	26.8	**18**	0.658	4.75	55.2
8	0.421	2.68	35.9	**19**	0.299	1.92	18.3
9	0.907	8.15	97.1	**20**	0.290	1.34	11.7
10	0.456	2.67	19.5	**21**	1.591	26.4	237.
11	0.173	0.550	6.78	**22**	0.195	0.890	6.92

6.3 Spilling Aggregation

A large cardinality microbenchmark. Spilly's TPC-H benchmark results are surprising because it stays comparatively fast when scaling beyond main memory capacity and even when spilling terabytes of data. We thus stress-test Spilly in the following experiment by executing a strongly out-of-memory microbenchmark that aggregates `lineitem` on 99% unique values using 88 Byte tuples:

```
select l_orderkey, l_partkey,
       min(l_shipinstruct), min(l_comment)
from lineitem
group by l_orderkey, l_partkey
```

Results. Figure 7 shows the tuple throughput on TPC-H SF 100 to 10,000. Spilly's throughput only decreases by a factor of 1.19× from SF 100 to 10,000, where it spills 5.3 TB of data. This is not because Spilly's baseline performance is bad; on the contrary, Spilly is faster than the other systems in both cases. We further investigate the missing performance cliff in the following.

6.4 Where is the Performance Cliff?

Tracing the aggregation microbenchmark. Given the small performance drop in Figures 6 and 7, one may suspect that Spilly's performance is limited by a different factor. We investigate this by tracing CPU utilization, memory bandwidth utilization, and I/O usage during the aggregation microbenchmark. Figure 8 shows these traces for SF 100 (top) and SF 1000 (bottom).

Performance analysis: In-Memory. On scale factor 100, the aggregation does not spill. While pre-aggregating and materializing data, it is CPU-bound, even while scanning from SSD (Figure 8 top left). After materialization, while building a shared hash table and aggregating into it, the query is memory bound (Figure 8 top right).

Performance analysis: Out-of-Memory. On scale factor 1000, the aggregation query starts spilling after a brief in-memory phase. While spilling, it writes with near-maximum I/O bandwidth, and almost fully utilizes the available memory bandwidth (Figure 8 bottom center and bottom right). Hybrid spilling gradually spills more partitions, resulting in a ramp-up phase where the full write throughput cannot be achieved. When reading back the spilled data, Spilly almost fully utilizes the available CPUs and memory bandwidth, meaning it is not I/O-bound while reading either.

Takeaway. To summarize, Spilly is only I/O bound in the write phase, and even there CPU and memory bandwidth utilization are near their maximum. Both the in-memory and out-of-memory queries are mostly limited by CPU and memory bandwidth, not I/O. This explains the missing performance cliff, and, more importantly, shows that large NVMe arrays are fast enough to be used for high-performance query processing without losing much performance. Section 6.8 shows how compression can help in scenarios where Spilly is more I/O-bound, e.g., when using fewer SSDs for spilling.

6.5 Hybrid Spilling

Impact of hybrid spilling on spilled data size. Section 4.3 describes how Umami generalizes the spilling approach applied by the hybrid hash join and makes it applicable to other operators. We call this generalized approach "hybrid spilling". To better understand

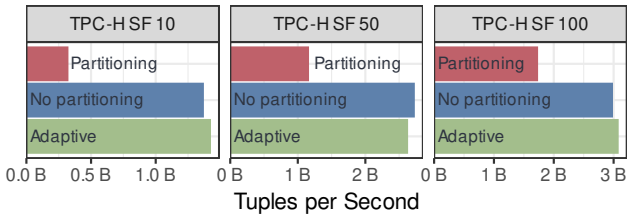

Figure 9: TPC-H performance with partitioning, non-partitioning, and adaptive operators. Data resides in memory.

how much hybrid spilling impacts performance, we compare it to a naiver approach that spills all materialized data if the engine runs out of memory, i.e., the approach of a spilling, non-hybrid grace join. The following table compares the spilled data size (left) and execution times (right) of both approaches on TPC-H (cold runs):

Scale Factor	Spilled (GB)			Response Time (s)		
	Spill All	Hybrid	×	Spill All	Hybrid	×
100	0	0	-	2.4	2.3	**1.01**
200	144	4	**36.64**	12.4	5.1	**2.42**
500	354	129	**2.74**	25.0	17.1	**1.46**
1,000	704	409	**1.72**	47.4	34.5	**1.37**
2,000	1,403	1,007	**1.39**	86.8	65.3	**1.33**
5,000	3,501	3,116	**1.12**	197.0	184.0	**1.07**
10,000	6,998	6,614	**1.06**	390.0	388.0	**1.01**

Results. As Section 6.5 shows, the benefit of Umami's hybrid spilling approach is largest when the materialized data slightly exceeds main memory capacity (384 GB): On SF 200, this approach spills 36× less data than the naive approach, though this advantage diminishes at larger scale factors. Since spilling in Spilly is fast, a 36× reduction in spilled data does not improve speed proportionally, particularly for large data sizes. Consequently, like the HHJ, hybrid spilling can smooth the performance cliff, but it does not improve the performance of extreme out-of-memory workloads.

6.6 Adaptive Materialization

Adaptive materialization in memory. The previous experiments show that Spilly is competitive with state-of-the-art in-memory engines and capable of out-of-memory query processing. However, we have not yet evaluated whether Umami's adaptive materialization degrades the in-memory performance of the simple hash join and non-partitioning aggregation. We thus repeat the motivational experiment shown in Figure 2 to see whether Umami improves the state of the art. Spilly either (1) always partitions, (2) never partitions, or (3) adaptively partitions (the default) and executes non-spilling TPC-H scale factors. The database is preloaded into memory, thereby minimizing factors such as scan performance.

Results. As Figure 9 shows, adaptive materialization drastically improves in-memory performance over partitioning variants while also staying out-of-memory-capable. We find that adaptive materialization causes less time to be spent in the kernel, e.g., for allocations. On the other hand, it also does not diminish performance compared to a simple hash join on average. Adaptive materialization sometimes even improves performance, which may be caused by improved locality through partitioning (c.f., Section 5.3), but the runtime adaptivity makes this difficult to determine.

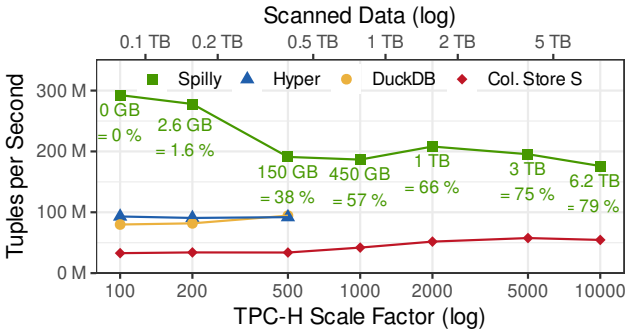

Figure 10: Join microbenchmark tuple throughput and Spilly spill size. Data is scanned from SSD (XFS + RAID-0).

Cost of hashing. During materialization, operators need to pass a hash to the Umami interface. Operators that employ pre-aggregation or HyperLogLog sketches compute this hash anyway, so applying Umami does not introduce overhead. Operators that do not already compute a hash, like a non-partitioning hash join not using HyperLogLog sketches, may experience overhead when introducing Umami. We measure this overhead by running the join microbenchmark (c.f. Section 6.7) in a modified version of Spilly that does not employ HyperLogLog sketches and passes a fake hash of 0 to Umami. The following table shows the instructions and cycles per tuple during the materialization phase of the join on TPC-H SF 100:

	Payload Bytes	Cycles/Tuple	Instr./Tuple	Time
No Hashing	199	1815.6	58.7	208.1 ms
Hashing	199	1827.1	82.2	209.1 ms
No Hashing	0	120.3	17.2	17.7 ms
Hashing	0	121.2	42.0	17.3 ms

Hashing introduces more instructions per tuple, but these are overshadowed by the loads and stores required during materialization and barely impact the cycles and time spent. Even for small tuples without a payload, where hashing more than doubles the instructions per tuple, the effect on the materialization time is small.

6.7 Spilling Join

A large join microbenchmark. After executing a spilling aggregation microbenchmark in Section 6.3, let us now also evaluate Spilly's join operator. To produce a large out-of-memory workload, this microbenchmark joins the `lineitem` and `partsupp` tables. It forces Spilly to spill and output 284 byte tuples by selecting large `char` and `varchar` columns:

```
select l_orderkey, l_shipinstruct,
       l_comment, ps_comment
from lineitem, partsupp
where ps_suppkey=l_suppkey and ps_partkey=l_partkey
```

Results. Figure 10 shows the resulting tuple throughput as the data size increases. Spilly's throughput decreases by a factor of 1.63× from scale factor (SF) 100 to 10,000, where it spills 6.5 TB of

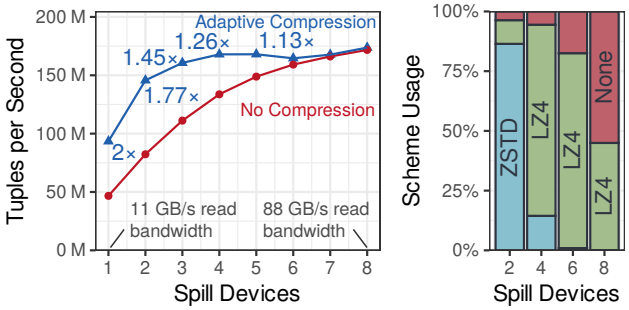

Figure 11: Performance (left) and compression scheme choice (right) with different NVMe array sizes. Data resides on SSD.

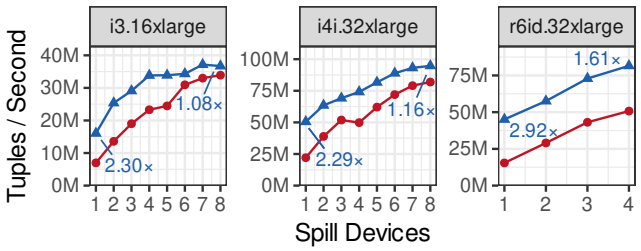

Figure 12: TPC-H Performance of Spilly on AWS EC2 instances; varying SSD counts, with and without compression.

data. Performance decreases most between SF 100 and 500, which is where Spilly starts spilling data. At larger scale factors, per-tuple performance remains nearly constant, independent of the amount of spilled data.

Performance of other systems. Figure 10 also shows DuckDB, Hyper, and Column Store S performance for reference. DuckDB and Hyper perform significantly better than Column Store S in this experiment, but fail to execute the query for scale factors larger than 500. Although DuckDB recently added support for spilling joins [1], we could not get it to scale further on our test machine. Column Store S did not utilize the full CPU capacity, memory bandwidth, or I/O bandwidth during the experiment on any scale factor. This explains the low but consistent performance of Column Store S and supports our theory that current out-of-memory-capable systems implement spilling techniques from the HDD era.

6.8 Self-Regulating Compression

Experimental design. While we demonstrated that Spilly can efficiently utilize an array of 8 PCIe 5.0 NVMe SSDs, few systems have access to such a large I/O bandwidth. The goal of self-regulating compression is to maximize the performance of spilling queries even for systems with fewer SSDs. To assess its effectiveness, we measure the performance of Spilly while changing the number of NVMe SSDs available for spilling. We compare this to a build of Spilly with self-regulating compression disabled via a compile-time flag, to also determine any runtime overhead introduced by the throughput tracking mechanisms (cf., Section 5.1). We execute the aggregation microbenchmark query from Section 6.3 on TPC-H SF 1000 data. Without compression, this query spills 457 GB.

Performance results. Figure 11 (left) shows the measured tuple throughput while varying the number of SSDs. Self-regulating compression consistently improves throughput, yielding a 2× speedup when only using one PCIe 5.0 SSD. As expected, this speedup decreases when using more SSDs to spill: With more than 6 PCIe 5.0 SSDs (≙ 39 GB/s write bandwidth), compression does not improve query performance further. Perhaps more importantly, it does not *decrease* performance in any scenario, showing that (1) it correctly disables compression when appropriate and (2) its throughput tracking mechanism has no significant runtime overhead.

Behavior of the optimization algorithm. As Figure 11 (right) shows, the optimization algorithm chooses ZSTD when little I/O bandwidth is available, and phases it out in favor of LZ4 and no compression as the number of SSDs increases. Thus, queries that have leftover CPU time can effectively utilize self-regulation compression to improve query latency. Furthermore, we would like to point out that self-regulating compression may also be useful for distributed query processing, where the available CPU and network I/O bandwidth can differ significantly depending on the setup.

6.9 Spilling in the Cloud

NVMe in the cloud. Cloud providers offer rentable virtual cloud instances with NVMe SSDs: AWS, for example, offers the *i3.16xlarge* and the *i4i.32xlarge* instances, each providing 8 NVMe SSDs [7, 8]. Each SSD offers roughly 2 GB/s read and 1 GB/s write bandwidth, allowing for up to 16 GB/s (8 GB/s) read (write) throughput on the entire NVMe array. For comparison, our experimental setup has 6× as much I/O bandwidth, but only 1.5× as many CPU cores as the newer (*i4i*, *r6id*) instance types. To test Spilly in such a different environment, we execute the spill-heavy aggregation microbenchmark on the aforementioned cloud instances while also varying the amount of SSDs available for spilling as in Section 6.8.

Results. As Figure 12 shows, self-regulating compression is always useful, and even more effective than on our on-premise setup on all cloud instances due to the higher CPU-to-I/O ratio. Interestingly, the best-performing instance (*i4i.32xlarge*) is only 1.5× slower than our server on this benchmark despite having 6× less I/O bandwidth. The *r6id* instance has as many CPU cores as *i4i* but fewer SSDs, limiting its performance on this large spilling query. The *i3* instance, which is cheapest in terms of SSD-bandwidth per dollar, does not benefit as much from self-regulating compression because it has fewer and older CPU cores.

7 CONCLUSION

This paper introduces Umami, an interface that provides adaptive materialization and self-regulating compression to enable efficient out-of-memory processing without sacrificing in-memory performance. We show that a query engine based on Umami can compete with state-of-the-art in-memory engines such as Hyper and DuckDB while also achieving the scalability and robustness of classical out-of-memory-capable systems. Additionally, we demonstrate that modern NVMe SSDs are capable of supporting high-performance query processing with near-in-memory performance when used correctly. Using Umami, we execute TPC-H on 10 TB of data on a server with only 384 GB main memory while keeping 89% of in-memory tuple throughput. This invalidates the common assumption that only in-memory query processing can be fast.

ACKNOWLEDGMENTS

Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] August 24, 2022. DuckDB Out-of-Memory Hash Join. https://github.com/duckdb/duckdb/pull/4189.
[2] January 16, 2023. DuckDB Benchmarking Guidelines. https://duckdb.org/faq#i-benchmarked-duckdb-and-its-slower-than-some-other-system.
[3] January 19, 2023. AWS c7g instance family. https://aws.amazon.com/de/ec2/instance-types/c7g/.
[4] January 19, 2024. https://rocksdb.org/.
[5] January 26, 2023. Tableau Hyper API. https://tableau.github.io/hyper-db/docs/.
[6] June 11, 2024. https://geizhals.de/micron-rdimm-32gb-mtc20f2085s1rc48ba1-a3017802.html.
[7] June 17, 2024. AWS i3 instance family. https://aws.amazon.com/ec2/instance-types/i3/.
[8] June 17, 2024. AWS i4i instance family. https://aws.amazon.com/ec2/instance-types/i4i/.
[9] March 26, 2024. https://www.scylladb.com/.
[10] November 1, 2023. Elevate Performance with 5th Gen Intel Xeon Processors Featuring Intel Accelerator Engines. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2023-11/5thgen-accelerator-engines-eguide.pdf.
[11] November 1, 2023. Intel 5th Generation Xeon Benchmarks. https://edc.intel.com/content/www/us/en/products/performance/benchmarks/5th-generation-intel-xeon-scalable-processors/.
[12] November 16, 2023. https://geizhals.de/micron-5100-pro-960gb-mtfddak960tcb-1ar16abyy-a1562532.html.
[13] November 16, 2023. Kioxia CM-7 Price History. https://www.idealo.de/preisvergleich/OffersOfProduct/203225197_-cm7-r-3-84tb-kioxia.html.
[14] November 16, 2023. Samsung PM1733 Price History. https://geizhals.de/samsung-ssd-pm1733-3-84tb-mzwlj3t8hbls-00007-a2202065.html.
[15] November 16, 2023. Samsung PM983 Price History. https://geizhals.de/samsung-ssd-pm983-3-84tb-mzqlb3t8hals-00007-a1870387.html.
[16] October 26, 2023. Skew-Aware Join in Postgres. https://github.com/postgres/postgres/blob/611806cd/src/include/executor/hashjoin.h#L95.
[17] September 13, 2023. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.
[18] September 26, 2023. DuckDB 0.9 Release Announcement. https://duckdb.org/2023/09/26/announcing-duckdb-090.html.
[19] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB*. Morgan Kaufmann, 169–180.
[20] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*. Morgan Kaufmann, 266–277.
[21] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. 2011. MaSM: efficient online updates in data warehouses. In *SIGMOD*. ACM, 865–876.
[22] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. 362–373.
[23] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD*. 168–180.
[24] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*. ACM, 37–48.
[25] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.
[26] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*. 54–65.
[27] John Cieslewicz and Kenneth A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*. 339–350.
[28] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*. 1–8.
[29] Jaeyoung Do and Jignesh M. Patel. 2009. Join processing for flash SSDs: remembering past lessons. In *DaMoN*. 1–8.
[30] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, and David J. DeWitt. 2013. Fast peak-to-peak behavior with SSD buffer pool. In *ICDE*. IEEE Computer Society, 1129–1140.
[31] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. 2011. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD Conference*. ACM, 1113–1124.
[32] Thanh Do, Goetz Graefe, and Jeffrey F. Naughton. 2022. Efficient Sorting, Duplicate Removal, Grouping, and Aggregation. *ACM Trans. Database Syst.* 47, 4 (2022), 16:1–16:35.
[33] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *PVLDB* 16, 11 (2023), 2769–2782.
[34] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*.
[35] Goetz Graefe. 2007. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*. 6.
[36] Goetz Graefe, Ross Bunker, and Shaun Cooper. 1998. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB*. 86–97.
[37] Goetz Graefe, Stavros Harizopoulos, Harumi A. Kuno, Mehul A. Shah, Dimitris Tsirogiannis, and Janet L. Wiener. 2010. Designing Database Operators for Flash-enabled Memory Hierarchies. *IEEE Data Eng. Bull.* 33, 4 (2010), 21–27.
[38] Jim Gray and Bob Fitzgerald. 2008. Flash Disk Opportunity for Server Applications. *ACM Queue* 6, 4 (2008), 18–23.
[39] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.
[40] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *PVLDB* 16, 9 (2023), 2090–2102.
[41] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*. ACM, 981–992.
[42] Sven Helmer, Thomas Neumann, and Guido Moerkotte. 2002. Early grouping gets the skew. *Technical reports* 2 (2002).
[43] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *PVLDB* 17, 3, 577–590.
[44] Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. 2022. Design Trade-offs for a Robust Dynamic Hybrid Hash Join. *PVLDB* 15, 10 (2022), 2257–2269.
[45] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. 195–206.
[46] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.
[47] Ioannis Koltsidas and Stratis Viglas. 2011. Data management over flash memory. In *SIGMOD*. 1209–1212.
[48] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *ICDE*. IEEE.
[49] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.
[50] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (2011), 298–309.
[51] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. 743–754.
[52] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.
[53] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.
[54] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proc. VLDB Endow.* 8, 10 (2015), 1058–1069.
[55] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. 3:1–3:8.
[56] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *SIGMOD*. ACM, 355–370.
[57] Aurosish Mishra, Shasank Chavan, Allison Holloway, Tirthankar Lahiri, Zhen Hua Liu, Sunil Chakkappen, Dennis Lui, Vinita Subramanian, Ramesh Kumar, Maria Colgan, Jesse Kamp, Niloy Mukherjee, and Vineet Marwah. 2016. Accelerating Analytics with Dynamic In-Memory Expressions. *PVLDB* 9, 13 (2016), 1437–1448.
[58] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD Conference*. 1123–1136.

[59] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. 1988. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *VLDB*. 468–478.
[60] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[61] Thomas Neumann and Viktor Leis. 2014. Compiling Database Queries into Machine Code. *IEEE Data Eng. Bull.* 37, 1 (2014), 3–11.
[62] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *DaMoN*. 7:1–7:9.
[63] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *CIDR*.
[64] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2018. Accelerating Concurrent Workloads with CPU Cache Partitioning. In *ICDE*. IEEE Computer Society, 437–448.
[65] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *CIDR*. www.cidrdb.org.
[66] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.
[67] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*. 755–766.
[68] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *DBTest*. 2:1–2:6.
[69] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. ACM, 1981–1984.
[70] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In *SIGMOD*. 1231–1242.
[71] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091.
[72] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates Disk-I/O Friendly Using SSDs. *Proc. VLDB Endow.* 6, 11 (2013), 997–1008.
[73] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD Conference*. ACM, 1961–1976.
[74] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *PVLDB* 8, 9 (2015), 934–937.
[75] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*. ACM, 1150–1160.
[76] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. 2009. Query processing techniques for solid state drives. In *SIGMOD*. 59–72.
[77] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *PVLDB* 16, 6 (2023), 1413–1425.
[78] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. 2022. What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!. In *ADMS*.
[79] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD* 29, 3 (2000), 55–67.
[80] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2021. Redy: Remote Dynamic Memory Cache. *PVLDB* 15, 4 (2021), 766–779.
[81] Zichen Zhu, Xiao Hu, and Manos Athanassoulis. 2023. NOCAP: Near-Optimal Correlation-Aware Partitioning Joins. *Proc. ACM Manag. Data* 1, 4 (2023), 252:1–252:27.
[82] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *ICDE*. 1349–1350.