What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!

Leonard von Merzljak
Technical University of Munich
leonard.von-merzljak@tum.de

Philipp Fent
Technical University of Munich
fent@in.tum.de

Thomas Neumann
Technical University of Munich
thomas.neumann@in.tum.de

Jana Giceva
Technical University of Munich
jana.giceva@in.tum.de

ABSTRACT

In the last ten years, SSDs achieved astonishing improvements in capacity per dollar and performance. Today, they are 30× cheaper than DRAM, and the difference is growing. Additionally, they are more than ten times faster than a few years ago, with a single SSD providing a throughput of 7 GB/s. Modern servers have enough PCIe lanes to directly attach multiple NVMe SSDs. That allows us to linearly scale the storage throughput and diminish the bandwidth gap between DRAM and SSDs. However, it requires a lot of parallel I/O requests to exploit multiple directly-attached SSDs, and the read latency is also very high.

In this paper, we propose to use asynchronous I/O and coroutines to continuously generate a lot of parallel I/O requests and hide the I/O latency. As a result, we get optimal throughput with up to 16× less compute resources than synchronous I/O, and we substantially flatten the performance cliff when exceeding main memory. We also show how to integrate coroutines into the code-generating, analytical DBMS Umbra and describe how we can call pre-compiled C++-Coroutines from the generated code. Finally, we present our new asynchronous index-nested-loop join algorithm that improves Umbra's end-to-end performance for analytical queries by up to 60%.

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/L-v-M/async.

1 INTRODUCTION

Traditional caching DBMSs [19] assume that memory is a scarce resource and that the primary location of the database must be on the slow disk. But decades of Moore's law have changed hardware fundamentally. In particular, DRAM became large and affordable enough to store significant fractions of most databases. The database community reacted by creating in-memory database systems [13]. Those systems redesigned many components of traditional caching systems to achieve orders of magnitude higher performance. For instance, Harizopoulos et al. [17] showed that even if they cache the entire database in-memory, the buffer manager is still the most expensive component of traditional systems. Since in-memory systems assume that the entire database fits into memory, they can substantially improve the performance by removing the buffer manager entirely.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). To Appear in the 13th Workshop on Accelerating Analytics and Data Management (ADMS22), September 2022, Sydney, Australia.

Table 1: Price and performance metrics of DRAM and SSDs.

	DRAM	SSD
config	8 × 64 GB	8 × 1.92 TB
cost-benefit	0.19 GB/$	5.8 GB/$
seq. read	152 GB/s (≥ 25 threads)	50 GB/s (≥ 4 threads)
rand. read	74 GB/s (≥ 72 threads)	48 GB/s (≥ 4 threads)
read. latency	181 ns (for 64 bytes)	73 µs (for 4 KiB)

1.1 In-Memory DBMSs Are Uneconomical

We currently observe two hardware trends that make us question the viability of pure in-memory systems and reconsider caching systems [26, 29]. First, the trend of rapidly dropping DRAM prices slowed down significantly in the last ten years [16]. Considering that the amount of data we want to analyze is ever-growing, it follows that the cost of buying sufficient memory capacity increases disproportionally. Therefore, in-memory systems are increasingly becoming uneconomical. Even though they provide the best performance, their cost/performance ratio can be worse than in caching systems [27].

1.2 The End of Slow Storage

The second hardware trend is that SSDs (i.e., NAND-based solid-state drives) have achieved astonishing improvements over the past years. Today, they are 30 times cheaper than DRAM and well on their way to catching up with magnetic disks in terms of capacity per dollar. The performance improvements are even more impressive. While the SATA interface only offered a bandwidth of 0.6 GB/s, modern SSDs are directly attached to PCIe using the NVMe interface and achieve a throughput of 7 GB/s and 1 million IOPS over 4 PCIe 4.0 lanes [2]. Faster standards are already specified and will soon offer up to 30 GB/s of throughput. With modern processors providing enough PCIe lanes for 16 directly-attached SSDs, the aggregated storage bandwidth approaches in-memory throughput while offering a much higher capacity at a fraction of the cost [16].

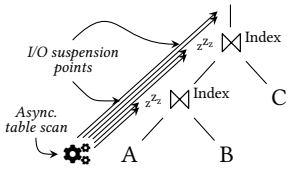

Figure 1: Bottom-up I/O parallel execution of a query.

In Table 1, we compare the price and performance of DRAM and SSDs in a server with two AMD EPYC 7713 CPUs with 64 cores each, 512 GiB of DDR4-3200 RAM, and 8 Samsung PM9A3 PCIe 4.0 NVMe SSD connected to a single socket. It shows that SSDs offer a 30 times higher capacity per dollar. While DRAM achieves a three times higher sequential read throughput of 152 GB/s, we also need at least 25 or six times as many threads as we need for SSDs to reach the maximum throughput of 50 GB/s. With four threads, on the other hand, we can only sequentially read 84 GB/s from DRAM. Additionally, for DRAM, the random read throughput is 50% lower than the sequential read throughput, and we need many threads to get there. For instance, with four threads, we can only read 9 GB/s randomly from DRAM, which is six times less than what we can read from SSDs with the same number of threads. Finally, the latency of randomly reading from DRAM is 400 times lower than from SSDs.

Such high storage throughput promises to eliminate the performance cliff when exceeding main memory. On the low end, this allows more economical high-performance data analysis (e.g., on a laptop), where we can rely on SSDs that are an order of magnitude cheaper than DRAM. On the high end, it enables single-node systems to process up to 100 TBs of data with near in-memory performance.

1.3 The Challenges of Fast Storage

Reaching this promised storage throughput is challenging for two reasons. First, it requires a lot of parallel I/O requests [14, 24, 30]. To achieve the maximum throughput, each SSD requires an I/O depth (i.e., the number of concurrent I/O requests) of at least 32, which we additionally have to multiply by the number of devices. Thus, we need several hundred parallel in-flight I/O requests, preferably even more. Current systems cannot generate so many parallel I/O requests or take too many computing resources to initiate them [11].

Second, the read latency of SSDs is still significantly higher than of main memory. If the system uses a synchronous interface for I/O requests, threads spend a substantial amount of time waiting for their completion. Meanwhile, both the CPU and the SSDs are underutilized.

1.4 Overview of the Paper

In this paper, we discuss how to combine asynchronous I/O with coroutines to significantly increase the storage throughput. We also integrated support for asynchronous I/O into our state-of-the-art, code-generating analytical DBMS Umbra. Our proposed system design uses asynchronous I/O to schedule hundreds of parallel I/O requests and hides I/O latencies with coroutines by suspending a function on an I/O request. As a result, we achieve higher I/O throughput with fewer threads. That frees up resources and allows us to increase the whole system throughput or switch to a more economical server.

We illustrate our envisioned system design in Figure 1. Our query processing uses the data-driven, push-based architecture that pushes tuples bottom-up through the execution plan [28]. The execution is driven by asynchronous table scans. Downstream operators receive hot in-memory data but might themselves need I/O, e.g., to find join-partners in an index. Until we have loaded the data, we suspend the pipeline on an I/O request and asynchronously process other tuples where the index pages might already be cached. Consequently, we generate a lot of parallel I/O requests and effectively exploit the SSDs.

The rest of the paper is structured as follows. Section 2 introduces the necessary background. In Section 3, we examine the benefits of asynchronous I/O and coroutines for query processing in a set of micro-benchmarks. Then, in Section 4, we discuss how to generate coroutines in a code-generating system and how those Codegen-Coroutines can call C++-Coroutines. Section 5 evaluates the end-to-end query performance of coroutine-based, asynchronous index-nested-loop joins in our compiling system Umbra. Finally, we review related work and summarize our contributions.

2 BACKGROUND

Database system design is a balancing act between storage, memory, and compute. In the last decade, we observed that SSDs improved faster than DRAM in terms of capacity per dollar and performance. Therefore, we argue that high-performance DBMSs should change how they use SSDs to become more economical.

2.1 Modern Caching DBMSs

LeanStore [26] is a high-performance buffer manager which uses pointer swizzling as a decentralized, low-overhead technique for address translation. The database system Umbra [29] builds on LeanStore and extends it with variable-size pages to simplify the implementation of buffer-managed data structures. Umbra's performance is comparable to the in-memory system Hyper [21] if the working set fits into memory and degrades gracefully for larger data sets. But Umbra uses synchronous I/O, which has undesired consequences and cannot fully exploit multiple, directly attached modern NVMe SSDs. Specifically, it blocks threads, restricts the amount of in-flight I/O operations, and demands high CPU utilization [16].

2.2 Asynchronous I/O to the Rescue

Asynchronous I/O solves our problems with synchronous I/O. It does not block threads, a single thread can schedule many I/O operations at once, and it reduces the CPU load [16].

In an asynchronous I/O interface, a worker thread submits a request for an I/O operation but does not wait for its completion. Instead, it continues to do other useful work and occasionally polls for a completion event. Once it receives this event, it can resume the operation that issued the I/O request.

Linux recently introduced a new interface for asynchronous I/O called `io_uring` [4]. It promises to be easier to use and provide better and more predictable performance than its predecessor `aio`.

2.3 Coroutines

Asynchronous I/O requires the ability to suspend and resume a function on an I/O request. Until recently, it was not easy to implement this in C++, which is the implementation language of Umbra. One approach was to transform functions into state machines [23], which decreases readability and, in turn, maintainability. Even worse, it requires a rewrite of large parts of the codebase.

Fortunately, with `C++20`, a simpler approach based on coroutines became available. By using special library types [6] and adding a keyword to a function, we can compile it into a coroutine which can suspend execution to be resumed later. That significantly decreases the effort to adapt an existing synchronous codebase to an asynchronous I/O interface.

2.4 Internal Parallelism of SSDs

We need asynchronous I/O to reach the high throughput of modern NVMe SSDs. Internally, SSDs contain dozens or even hundreds of flash chips that manage a subset of the storage cells and can be accessed in parallel [14, 24]. The SSD controller transparently distributes writes across the chips at page granularity. Therefore, when reading several pages simultaneously, we exploit multiple flash chips and achieve higher bandwidth. Asynchronous I/O enables us to schedule hundreds of parallel I/O requests and continuously provide work for all flash chips. Furthermore, asynchronous I/O coupled with direct I/O reduces the CPU load which is required to exploit multiple SSDs connected to a single machine [16].

3 ASYNCHRONOUS I/O FOR QUERY PROCESSING

Throughput and latency are two key metrics to measure the performance of storage. As shown in Figure 2, we can linearly scale the throughput until we run out of PCIe lanes by buying multiple SSDs and directly attaching them to PCIe using the NVMe interface. However, to exploit them, we also need to scale the amount of parallel I/O requests. But the read-latency of SSDs is 400× higher than that of DRAM, and, unfortunately, we cannot reduce this gap.

Asynchronous I/O and coroutines enable us to maximize the performance when reading from SSDs:

- Asynchronous I/O facilitates the scheduling of hundreds of parallel I/O requests to reach high throughput.
- Coroutines allow us to hide the latency by suspending a function on an I/O request and resuming another.

In this section, we micro-benchmark if asynchronous I/O and coroutines improve the performance of analytical data processing on SSDs. Our results show that asynchronous I/O considerably flattens the performance cliff when exceeding main memory. Additionally, it reduces the compute requirements to reach full SSD speed by about 4×.

3.1 Sequential I/O for Table Scans

We start with the simple case of sequential I/O for table scans. The SSD controller optimizes for this access pattern by transparently striping larger files across the flash chips at page granularity [24]. Consequently, large sequential scans take full advantage of the internal parallelism offered by SSDs, and even synchronous reads should achieve high throughput.

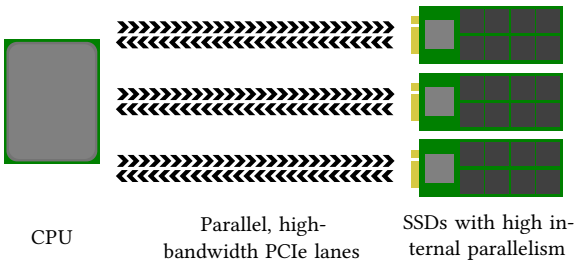

Figure 2: The storage architecture of modern SSDs.

We want to examine the performance of sequential reads on multiple directly-attached SSDs to check if there are benefits of asynchronous I/O over synchronous I/O. Therefore, we implemented query 1 of the TPC-H benchmark [10] by hand and used C++-Coroutines and io_uring for asynchronous I/O. Our implementation is similar to the hand-written one described by Boncz et al. [7].

Query 1 performs a table scan of the lineitem relation and applies a filter that removes less than two percent of the tuples. It then groups the remaining tuples into four groups and computes several simple aggregates on top of them.

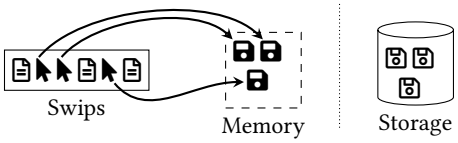

Figure 3: Swips directly address in-memory data (🖱) or reference data paged out to storage (🖹).

Since we are interested in the performance impact of having to read from storage, we implemented a static buffer manager that allows us to fix the fraction of cached pages before executing the benchmark. Similar to LeanStore [26] it consists of a vector of swips with one swip for each page of the lineitem relation. As shown in Figure 3, a swip is either a virtual memory address or a page identifier. We encode a swip as a 64-bit integer containing either a virtual memory address (i.e., a 48-bit pointer) or a 63-bit page identifier. Then we can use the topmost bit to discriminate between the two states (i.e., pointer tagging). If the swip contains a virtual memory address, the referenced page is already cached and directly accessible by dereferencing the address. Otherwise, we use the page identifier to locate and load the page from storage. Before a benchmark run, we decide how much data should be already cached and populate the buffer manager accordingly. For example, if we want 60% of the data cached, we choose 60% of the swips according to a uniform random distribution, load the referenced pages into memory, and change the swips to contain pointers to the cached pages.

For synchronous I/O, the number of threads directly determines the amount of parallel I/O requests. Therefore, we also parallelize and vary the parallelism in our micro benchmarks. For that, we use

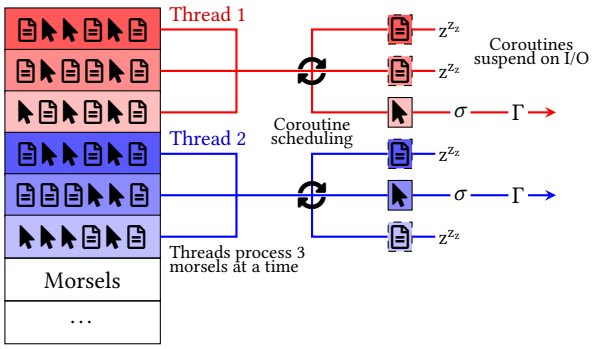

Figure 4: Threads fetch multiple morsels for table scans and start one coroutine per morsel.

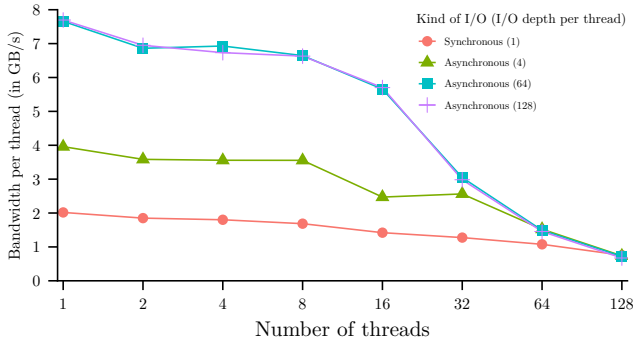

Figure 5: Throughput per thread of processing TPC-H Q1. Page size of 64 KiB, 60% cached.

morsel-driven parallelism with one worker thread per core, which scales well to many core execution [25]. During processing, threads repeatedly grab a morsel consisting of a few consecutive swips from the central work queue, filter the tuples on the referenced pages, and aggregate them in a thread-local hash table. After all pages are processed, a final merge step combines the intermediate thread-local results into a final result.

For the synchronous I/O version, the thread iterates over the morsel's swips. If the referenced page is not cached, the thread issues a blocking `pread()` call to load it and then processes its tuples. While waiting for the data, the thread is idle.

For the asynchronous I/O version, we use a coroutine-per-morsel approach, as shown in Figure 4. We configure the maximum I/O depth per thread, which corresponds to the number of morsels each thread picks at once to execute. The example in the figure uses an I/O depth of three, which means that each thread grabs three morsels from the central work queue and starts three coroutines.

If a coroutine encounters a page identifier within the current morsel, it issues an asynchronous, non-blocking read operation and cooperatively suspends itself. For cached, in-memory pages, the coroutine synchronously processes the tuples on them. As long as there are outstanding I/O operations and no ready coroutines, the thread polls the thread-local io_uring for newly completed I/O requests and resumes the coroutines that issued them. Only once the entire batch of morsels is finished does the thread grab the next batch from the work queue.

3.1.1 Higher Throughput with Less Compute. In the first micro-benchmark, we examine the effect of coroutines and asynchronous I/O on the throughput per thread while processing TPC-H Q1. With synchronous I/O, one thread can have at most one outstanding I/O request. Asynchronous I/O, on the other hand, allows one thread to issue many I/O requests simultaneously. But since I/O still consumes a lot of CPU cycles, one thread is not enough to saturate the I/O bandwidth [16].

Before starting a benchmark run, we configure the static buffer manager to use a fixed page size of 64 KiB and ensure that a random subset of 60% of the lineitem relation's pages are cached. For comparison, we vary the number of threads and the I/O depth per thread. While synchronous I/O only supports an I/O depth per thread of one, we test different asynchronous I/O versions with I/O depths per thread of up to 128.

We show the results of the micro-benchmark in Figure 5. It visualizes the throughput per thread (y-axis) of processing TPC-H Q1 for varying numbers of threads (x-axis) and different types of I/O (color). We used the hardware described in Table 1 with a maximum I/O throughput of 50 GB/s. But note that we are measuring the throughput of processing data from the lineitem relation for TPC-H Q1. Since 60% of it is already cached, we can achieve higher throughput than 50 GB/s.

For up to 64 threads, asynchronous I/O considerably outperforms synchronous I/O. For a single thread, asynchronous I/O with an I/O depth of 64 per thread achieves an almost four times higher throughput than synchronous I/O. Additionally, we see that an I/O depth of 64 per thread is required to get the best throughput with asynchronous I/O. Going higher does not further improve the throughput. Finally, starting with 64 threads, the difference between synchronous and asynchronous I/O diminishes. That is expected behavior as a higher number of threads automatically leads to a higher number of in-flight I/O requests. The result is higher throughput because of better utilization of the internal parallelism of the SSDs.

In summary, asynchronous I/O allows us to reach higher throughput than synchronous I/O with 1/4 the compute resources. For example, with four threads and an I/O depth per thread of 64, we achieve a total throughput of 27.7 GB/s which is more than the 22.7 GB/s we get with 16 threads and synchronous I/O. Similarly, with 16 threads and an I/O depth per thread of 128, we reach a total throughput of 91.2 GB/s which is more than the 68.9 GB/s we get with 64 threads and synchronous I/O. This frees up resources for in-memory workloads, or allows downsizing the compute resources for more economical operation.

3.1.2 Graceful Degradation. In the second micro-benchmark, we investigate the performance impact when the working set's size exceeds the memory capacity and check if asynchronous I/O and coroutines help soften it.

In Figure 6, we measure the throughput of processing TPC-H Q1 (x-axis) for different fractions of pages cached in main memory (y-axis). We compare synchronous I/O with asynchronous I/O having an I/O depth per thread of 128, which achieved the best performance

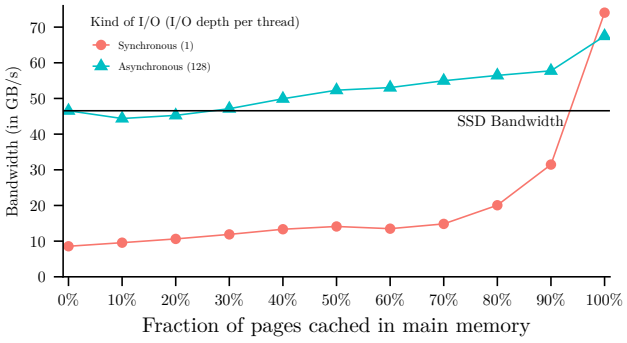

Figure 6: Throughput of processing TPC-H Q1. Page size of 64 KiB, 8 threads.

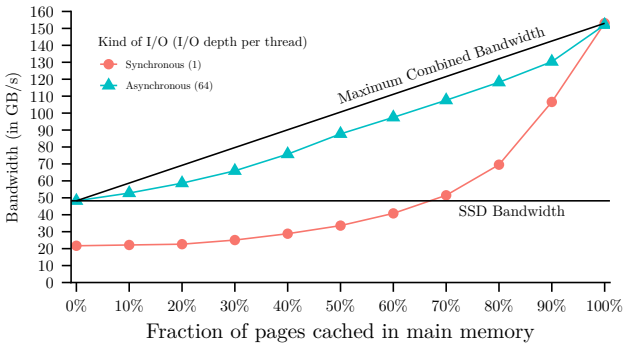

Figure 7: Throughput of processing TPC-H Q1. Page size of 64 KiB, 32 threads.

in this benchmark. Again we use a page size of 64 KiB, but this time fix the number of threads to eight. Furthermore, we add an auxiliary line to mark the bandwidth achieved with asynchronous I/O with an empty cache.

We see that eight threads are sufficient to achieve the full I/O bandwidth of 47 GB/s as long as we use asynchronous I/O. The I/O depth of synchronous I/O is too low, resulting in five times worse throughput. As we increase the fraction of pages cached in memory, the throughputs for both synchronous and asynchronous I/O only improve marginally. But even with 90% cached, the throughput of synchronous I/O is still 15 GB/s below the throughput of asynchronous I/O when nothing is cached.

For Figure 7, we repeat the experiment with 32 threads and add one more auxiliary line showing the theoretical optimal throughput based on the SSD and memory bandwidths. As expected, the increased number of threads improves the throughput for synchronous I/O. Still, asynchronous I/O outperforms synchronous I/O and gets very close to the optimal theoretical throughput. Interestingly, while eight threads were sufficient to reach the available SSD bandwidth, we needed 32 threads on our 64 core socket to saturate the DRAM bandwidth of 150 GB/s. That is in line with the numbers shown in Table 1.

To sum up, asynchronous I/O helps tremendously to reduce the impact of exceeding main memory. Especially for workloads that *almost* fit into main memory, e.g., with 80 % cached, synchronous I/O loses more than 50 % of the in-memory performance. In contrast, asynchronous I/O retains about 75 % of this performance.

3.2 Random I/O for Index Lookups

In the last section, we discussed that even sequential workloads have significant random access patterns, such that asynchronous I/O is very beneficial. We now turn to random reads in index structures, where the observed effects should be even more pronounced.

For SSDs, the throughput of random and sequential I/O is almost identical (c.f. Table 1). However, the latency penalty when randomly reading from SSDs is high and limits workloads that perform a lot of random I/O, such as index lookups. Therefore, we conducted a second set of micro-benchmarks to evaluate whether asynchronous I/O and especially coroutines improve the performance of queries with random I/O. For that, we implemented query 14 of the TPC-H benchmark by hand in C++ and again used C++-Coroutines and io_uring for asynchronous I/O.

Query 14 performs a very selective filter on the lineitem relation and joins the remaining tuples with the part relation on the foreign-key column. Finally, it aggregates the tuples using a single group.

For the experiment, we evaluate the join using the index-nested-loop algorithm. That is the same strategy that the Hyper system uses. Building the hash index for the part relation is done upfront before starting the benchmark. The hash index is implemented like the hash table in Leis et al. [25]. Each entry in the index maps the join key to a pair consisting of a swip and a tuple offset. Thus, every lookup in the index yields the exact location of the corresponding tuple. Additionally, while building the index, we compute how often each page of the part relation is going to be read during query execution. For a benchmark run, we use this knowledge to fix the fraction of accesses that should be a cache hit and load the required pages into memory beforehand. To isolate the effects of random I/O, we load the entire lineitem relation into memory before executing the micro-benchmarks as it is read sequentially.

We again parallelize the execution with morsel-driven parallelism and use the coroutine-per-morsel approach. Threads repeatedly grab a batch of morsels of in-memory lineitem tuples from the central work queue, filter them according to the predicate, and look up the join partners in the hash index. If the referenced part page is not cached, the thread loads it from storage using synchronous or asynchronous I/O with coroutines, respectively. For the latter, the coroutine responsible for the morsel suspends itself after issuing the I/O request, and the thread switches to another coroutine from the batch as already described above.

For the experiment, we execute query 14, fix the page size to 4 KiB, and prepare the static buffer manager to serve 60% of the lookups in the part relation index from the cache. Larger page sizes than 4 KiB significantly increased the response times but did not change the relative differences between synchronous and asynchronous I/O.

We display the results of the micro-benchmark in Figure 8. It shows that asynchronous I/O and coroutines consistently outperform synchronous I/O for workloads consisting of random reads. With a single thread, asynchronous I/O with an I/O depth per thread

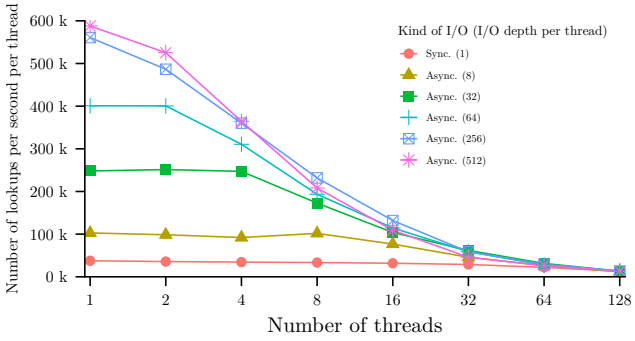

Figure 8: Lookups per second per thread of processing TPC-H Q14. Page size of 4 KiB, 60% cached.

of 512 achieves an almost 15 times higher throughput than synchronous I/O. Even with 32 threads, the throughput is still more than twice as high as with synchronous I/O. But the results are even more impressive when we compare the absolute numbers. With 128 threads, synchronous I/O reaches its maximum of 1,607,774 lookups per second. For asynchronous I/O, we only need four threads to surpass that number with 1,854,512 lookups per second. That means better performance with 16 times fewer threads.

In conclusion, the results show that we need to use asynchronous I/O and coroutines if we want to exploit the performance of modern NVMe SSDs. Furthermore, asynchronous I/O frees up resources which enables us to complete more work with the same hardware or the same work with much cheaper hardware.

4 COROUTINES IN A CODE-GENERATING DBMS

In the last section, we have demonstrated that asynchronous I/O has enormous benefits for query processing. It is significantly more efficient than synchronous I/O and enables graceful performance degradation when exceeding main memory. But to use it, we need coroutines and an asynchronous execution model, which is challenging to integrate into a system design. Interpreting query engines should be able to use coroutines straightforwardly. But those engines have the inherent interpretation overhead. Code generating systems are more efficient but require higher engineering effort to support coroutines if the compilation target does not offer them (e.g., C code or LLVM IR). In this section, we thus develop a novel system to generate asynchronous code and show how to integrate it into a state-of-the-art code-generating DBMS.

4.1 C++-Coroutines

A coroutine is a function that can suspend execution at well-defined suspension points to be resumed later. This way, we can write sequential code that executes asynchronously (e.g., to handle asynchronous I/O) [1]. Support for coroutines is one of the language features added to C++20. Since our database system, Umbra, is written in C++, we can use them for asynchronous I/O. However, Umbra is a code-generating system, and the code it generates to evaluate a query is not in C++. Instead, our code generator translates a query plan into a custom intermediate representation, Umbra IR, and we have several compilation backends that lower Umbra IR into executable code [22]. For example, the flying start backend emits x86 machine code, and the LLVM backend transforms Umbra IR into LLVM IR for compilation with LLVM [28]. Nevertheless, we have a proxy system in Umbra that makes it possible to call pre-compiled C++ code, even coroutines, from the generated code of a query.

4.2 Codegen-Coroutines

In this section, we present Codegen-Coroutines. When we generate code for a coroutine in Umbra, our compilation backends translate this code into a state machine similar to how a C++-Compiler compiles C++-Coroutines. First, we explain how this translation works and how we can await C++-Coroutines from a Codegen-Coroutine. Finally, we discuss how we use C++-Coroutines for asynchronous I/O.

A coroutine is a generalization of a function. Next to the usual `call` and `return` operations, coroutines additionally provide `suspend` and `resume` [5]. Suspending a coroutine means stopping its execution at a well-defined suspension point and transferring control back to the caller or resumer. The activation frame, which holds the coroutine's state (e.g., parameters and local variables), is not destroyed. As a result, all objects in scope at the suspension point remain alive. When we resume a suspended coroutine, its activation frame becomes active again. In this case, the coroutine resumes execution immediately after the last suspension point.

Listing 1: A simple C++-Coroutine.

```
task<int> read64(int numInts) {
  int bytesRead = co_await read8(numInts * 8);
  co_return bytesRead / 8;
}
```

In Listing 1, we show a C++ function that uses the `co_await` and `co_return` operators. By definition, the C++-Compiler compiles such a function into a coroutine. Each usage of the `co_await` operator marks a potential suspension point. In this example, we show a coroutine that reads integers by *awaiting* another coroutine, `read8()`, that reads several bytes. If the *awaited* coroutine (`read8()`) does not suspend and finishes synchronously, the *awaiting* coroutine (`read64()`) does not suspend as well. Otherwise, both coroutines suspend, and the *awaiting* coroutine becomes the *continuation* of the *awaited* coroutine. When the *awaited* coroutine is later resumed and runs to completion, it will automatically resume the *awaiting* coroutine. Note that the coroutine returns an object of type `task<int>`, which has special meaning for the compiler and allows library writers to customize the behavior of the coroutine.

For illustration, let's say we want to generate code for a Codegen-Coroutine that calls the C++-Coroutine of Listing 1 and returns a computed value. It should perform the same computation as the C++-Coroutine shown in Listing 2.

As mentioned above, our compilation backends do not generate C++ code but instead transform a Codegen-Coroutine into a state machine. To ease the explanation, we show the code emitted by our C compilation backend, which is only used internally for debugging.

For each Codegen-Coroutine, our C backend generates a (1) coroutine state, a (2) `resume` function, and a (3) `ramp` function.

Listing 2: A C++-Coroutine similar to the Codegen-Coroutine we want to generate.

```
task<int> readTuple(int numTuples) {
  int c = co_await read64(numTuples);
  int d = co_await read8(numTuples);
  co_return c + d;
}
```

The coroutine state is the *activation frame* of the coroutine. We separate it into two types (cf. Listing 3). All Codegen-Coroutines use an object of the general type `CoroState` to hold internal variables required to manage the execution of the coroutine. `ReadTupleState` is specialized for the specific coroutine we want to generate and provides storage for parameters and local variables whose values we must preserve across suspension points.

Listing 3: Coroutine states are split into a general and a specialized type.

```
struct CoroState;
typedef CoroState* (*ResumeFunc) (CoroState*);

// General type
struct CoroState {
  ResumeFunc resumeFunc;
  int currentState;
  CoroState* continuation;
  int returnValue;
};

// Specialized type
struct ReadTupleState {
  CoroState coroState;
  int numTuples, c;
  void* coroHandle;
}
```

Before we show the code for the generated `resume` and `ramp` functions of our Codegen-Coroutine, we must discuss how we can *await* C++-Coroutines from a Codegen-Coroutine. We achieve this by wrapping the C++-Coroutine with two separate functions we write by hand and ship with the system. The first, shown in Listing 4, starts the execution of the C++-Coroutine by calling the `awaitSuspend()` method of the `task` object. This method returns the information if the C++-Coroutine has finished synchronously with a handle to the C++-Coroutine. Note that we don't delete the *activation frame* yet, even if it finished synchronously, as we still need to extract the result of the coroutine later in the second function. Therefore, we call the `release` method of the `task` object to release its ownership of the *activation frame*. If the C++-Coroutine did not finish synchronously, the `awaitSuspend()` method sets the Codegen-Coroutine to be the *continuation* of the C++-Coroutine.

The second function to wrap a C++-Coroutine is shown in Listing 5. It is called after the C++-Coroutine has run to completion and uses the *coroutine handle* obtained earlier to extract its result

Listing 4: The first wrapper function starts the C++-Coroutine.

```
struct Result {
  bool shouldSuspend;
  void* coroHandle;
};

Result proxyRead8(int numInts,
                  const CoroState* continuation) {
  task<int> t = read8(numInts);
  bool shouldSuspend = t.awaitSuspend(continuation);
  void* coroHandle = t.release().address();
  return {shouldSuspend, coroHandle};
}
```

(the value returned by `co_return`). It does this by constructing a new `task` object that takes ownership of the existing *coroutine handle*. Furthermore, the destructor of this `task` object also deletes the activation frame associated with the *coroutine handle*.

Listing 5: The second wrapper function extracts the result of the C++-Coroutine and deletes its activation frame.

```
int extractRead8Result(void* coroHandle) {
  return task<int>{coroHandle}.result();
}
```

Next, we show the generated `resume` function in Listing 6. This function contains the actual state machine and represents the body of our Codegen-Coroutine. It uses the `currentState` member of its coroutine state to remember the last suspension point and, thereby, the location in the code where it should resume execution.

The resume function uses the two wrapper functions described above to call the C++-Coroutines. In the initial state 0, it calls the first coroutine `proxyRead64()` to start the execution and set itself as the *continuation* of the C++-Coroutine. Then, it saves the returned coroutine handle in its state to be able to access it across suspension.

If the C++-Coroutine did not finish synchronously, the Codegen-Coroutine prepares itself for suspension by updating its `currentState` and returns a `NULL` pointer to signal to its caller or resumer that it is suspended. Otherwise, the Codegen-Coroutine continues and extracts the result of the C++-Coroutine by calling the second wrapper function. When the Codegen-Coroutine runs to completion, it returns the value of its coroutine state's `continuation` member.

This simple example also illustrates the inherent overhead of using coroutines. Calling a coroutine for a cheap function has high overhead. We even have to allocate the coroutine state on the heap, which further increases the overhead. But since all coroutine states of a specific coroutine have the same size, we can write an efficient allocator for that. For expensive functions that perform I/O, the overhead of calling a coroutine is more than compensated for by the increased I/O parallelism.

Finally, Listing 7 shows the `ramp` function, which initializes the coroutine state and transfers control to the `resume` function. It is the entry point of our Codegen-Coroutine and is called only once.

Listing 6: The generated `resume` function of our Codegen-Coroutine.

```
CoroState* resumeReadTuple(ReadTupleState* state) {
switch (state->coroState.currentState) {
case 0: {
  // co_await read64
  Result r = proxyRead64(state->numTuples,
                         state->coroState);
  state->coroHandle = r.coroHandle;
  if (r.shouldSuspend) {
    state->coroState.currentState = 1;
    return NULL;
  }
} // fall through
case 1: {
  // resume from read64
  state->c = extractRead64Result(state->coroHandle);
  // co_await read8
  Result r = proxyRead8(state->numTuples,
                        state->coroState);
  state->coroHandle = r.coroHandle;
  if (r.shouldSuspend) {
    state->coroState.currentState = 2;
    return NULL;
  }
} // fall through
case 2: {
  // resume from read8
  int d = extractRead8Result(state->coroHandle);
  state->coroState.returnValue = state->c + d;
  // finished execution, resume the continuation
  return state->coroState.continuation;
}
}
}
```

The `resume` function, on the other hand, is called as often as needed to resume a suspended coroutine after a suspension point until the coroutine runs to completion.

Listing 7: The generated `ramp` function of our Codegen-Coroutine.

```
CoroState* rampReadTuple(ReadTupleState* state,
                         CoroState* continuation,
                         int numTuples) {
  state->coroState.resumeFunc = &resumeReadTuple;
  state->coroState.currentState = 0;
  state->coroState.continuation = continuation;
  state->numTuples = numTuples;
  return resumeReadTuple(state);
}
```

We now have all the building blocks required to generate code for coroutines and call pre-compiled C++-Coroutines from Codegen-Coroutines. This allows us to extensively use C++-Coroutines in Umbra and, e.g., build an asynchronous B+-tree index in C++ that we can easily use from the generated code.

4.3 Asynchronous I/O

An asynchronous interface such as io_uring offers two fundamental operations: submitting a request and polling for the completion of said request. Listing 8 shows the C++-Coroutine we use to submit an asynchronous read request. The `IOUringReadAwaiter{}` is a so-called *awaiter* that contains code to submit the I/O request to io_uring, store a handle to the coroutine with the request (see `std::coroutine_handle<Promise>::address`), and then suspend the coroutine. We provide another function to poll for the completion of the requests. If there are any completed requests, we extract the coroutine handles from them and resume the corresponding coroutines from their last suspension points by calling the `resume()` method of the coroutine handle.

Listing 8: The C++-Coroutine we use to submit an asynchronous read request.

```
task<bool> doAsyncRead(IOUring& ring, int fd,
                       void* buf, size_t count,
                       off_t offset) {
   while (count) {
      auto res = co_await
        IOUringReadAwaiter{ring, fd, buf, count,
                           offset};
      if (res < 1) {
         co_return false;
      }
      count -= res;
      buf = static_cast<char*>(buf) + res;
      offset += res;
   }
   co_return true;
}
```

In the next section, we provide more details on how we use the building blocks presented here to enable asynchronous I/O. Thanks to the building blocks, we can realize an asynchronous execution model in our code-generating DBMS Umbra. And since we reuse our existing machinery for calling pre-compiled C++ code, calling a C++-Coroutine is just as simple as calling a normal C++ function.

5 ASYNCHRONOUS I/O AND COROUTINES IN UMBRA

In Section 3, we discovered that asynchronous I/O with coroutines significantly and consistently outperforms synchronous I/O for random reads. In this section, our goal is to reproduce those results in Umbra by integrating asynchronous I/O and Codegen-Coroutines to improve the end-to-end performance of analytical queries. Following the insights of Section 3, we focused on random I/O and integrated asynchronous I/O into our index-nested-loop join algorithm.

Query 5 of the TPC-H benchmark is a good example of a query for which Umbra's optimizer chooses an index-nested-loop join

algorithm. It joins a small intermediate result with the 130 times larger lineitem relation and the join key is a prefix of the lineitem relation's primary key. Since Umbra maintains a B-Tree index for primary keys, we can use the index-nested-loop algorithm to efficiently evaluate the join. The benefit of doing so is that we don't need to read the entire lineitem relation but only the pages relevant for the join. As the access pattern is inherently random, we expect that coroutines, with their ability to hide the I/O latency and maximize the I/O parallelism, should give a noticeable performance boost to this evaluation strategy.

Changing the code to use coroutines and asynchronous I/O was straightforward. First, we needed to adapt the C++ code for the B-Tree and page-lookup to use C++-Coroutines. For that, we had to change the return types from `T` to `task<T>` of all functions on the path down to the ones that read from storage. Additionally, we had to use the `co_await` operator every time we called such a function to compile the caller into a C++-Coroutine. Finally, we changed Umbra's code generator to compile the generated functions responsible for the asynchronous lookups into Codegen-Coroutines.

Our first attempt was to start one Codegen-Coroutine for each tuple of the probe side to maximize the I/O parallelism. When the thread-local io_uring instance became full, the thread started to poll the io_uring for completion events and resumed the suspended coroutines accordingly. However, it turned out that the overhead of allocating a coroutine state, materializing the tuple in it, and checking if the io_uring still has free capacity is too high for only a single tuple. Instead, we experimentally verified that we must batch at least 128 tuples in a Codegen-Coroutine to amortize the overhead of initializing and starting it.

The second problem we encountered was that when we have a lot of concurrent I/O requests, it often happened that multiple coroutines tried to load the same page simultaneously. For correctness reasons, we cannot have multiple copies of the same page cached in the buffer manager. Also, it is obviously more efficient to schedule the I/O request just once. To avoid this, we mark a swip as locked in Umbra while loading the corresponding page into memory and use our parking lot infrastructure [8] to let other threads wait for the in-flight I/O operation. With coroutines, it is necessary to adapt the parking lot infrastructure since the condition variable used internally would block not only the coroutine but also the entire thread. We have left those adaptions for future work. Instead, we queue those coroutines in a round-robin scheduler and use a restarting/retrying mechanism to prevent the thread from blocking.

5.1 Evaluation

The micro-benchmarks presented in Section 3 assume an idealized system with a simplistic buffering strategy. We now examine how the findings generalize to the challenges of a real system with a complex buffer manager, MVCC checks and full SQL support. For the evaluation, we use TPC-H queries 4, 5, and 10, for which Umbra's optimizer picks our new asynchronous index-nested-loop join algorithm described above. The TPC-H database has a scale factor of 100, corresponding to 100 GB of raw data. We conducted all measurements on a server with two AMD EPYC 7713 CPUs with 128 cores, 512 GiB of DDR4-3200 RAM, and 8 Samsung PM9A3 PCIe 4.0 NVMe SSD connected to a single socket. During the benchmarks, we pinned the process to the same socket to which the SSDs are connected. However, we did not see significantly different performance for cross-socket PCIe reads. The code was compiled using GCC 11.2.

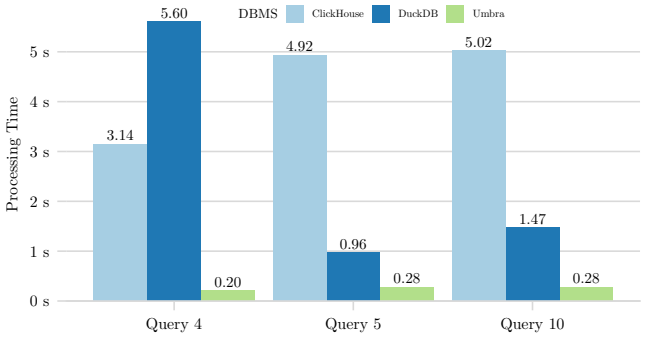

Figure 9: Systems comparison on the TPC-H benchmark.

5.1.1 Systems Comparison. Before we look at the performance of our new asynchronous index-nested-loop join, we want to understand how fast Umbra executes TPC-H queries 4, 5, and 10 compared to other state-of-the-art analytical DBMSs. The significance of this comparison is that it is much harder to improve the performance of a very fast system than that of a slow system. We compare Umbra with ClickHouse v22.6.1.1985-stable [9] and DuckDB v0.4.0 [33]. Since ClickHouse does not support external index-nested-loop joins, it only makes sense to compare the raw processing performance of the different systems. Therefore, we give each system the full 512 GiB of DRAM and use a schema without any indexes so that the systems have to perform full table scans. We configure the systems to use 64 threads on our 64-core socket. Other than that, we use the default configuration. Additionally, we preload the data into memory and report the median of 10 runs. Since ClickHouse's query optimizer does not perform join-reordering, we write the queries to use the join order picked by Umbra's optimizer.

We show the results in Figure 9. They show that Umbra considerably outperforms the other systems and is 15, 3.5, and 5.35 times faster than the next best system.

5.1.2 Asynchronous Index-Nested-Loop Joins. We now examine the asynchronous index-nested-loop joins described earlier in the section. For the evaluation, we adjusted Umbra to use 16 threads, an I/O depth per thread of 256, and to batch 128 tuples per coroutine. Furthermore, we configured Umbra to use direct I/O and varied the size of the buffer manager in the benchmark runs. Since we used a TPC-H scale factor of 100, 20% cached, e.g., translates to a buffer manager size of 20 GB.

We visualize the results of our experiments in Figure 10. The results show that asynchronous index-nested-loop joins considerably improve the performance of the three queries. For a buffer manager size of 20 GB, asynchronous I/O improves the performance of query 4 by almost 60%. Even though Q5 and Q10 also execute other operators, asynchronous index-nested-loop joins still improve the

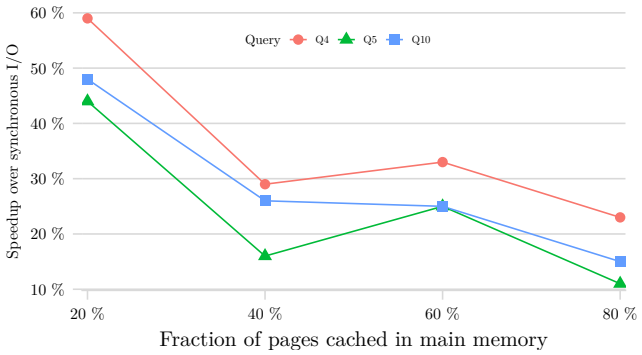

Figure 10: Asynchronous index-nested-loop joins on the TPC-H benchmark.

performance by at least 45%. With more data cached, the improvements unsurprisingly become smaller. But with 80% cached, Q4 is still almost 25% faster than the version with the synchronous index-nested-loop join.

When we increase the number of threads to 64 or 128, the performance difference between synchronous and asynchronous I/O disappears. That is in line with our insights from Section 3 that an increasing number of threads automatically leads to higher I/O parallelism and better throughput.

5.1.3 Conclusion. Although we have only added support for asynchronous I/O to our index-nested-loop join algorithm, we can already see considerable performance improvements for out-of-core joins. In future work, we also want to add asynchronous I/O to our table scan operators to realize our vision of a system based on asynchronous execution that can exploit the performance of multiple NVMe SSDs.

6 RELATED WORK

Many traditional DBMSs use the pull-based iterator model [15], which is not well suited for asynchronous execution. Newer cloud-native systems, e.g., Snowflake [12] or Redshift [3], instead use the push-based execution model and can immediately pass tuples to the next operator when they are read from the storage or network. Push-based systems are well suited for asynchronous execution. However, we are unfortunately not aware of any publications directly discussing the benefits of asynchronous I/O for query processing.

There is already a plethora of research from the field of in-memory systems showing that one can use C++-Coroutines to hide the CPU-cache miss latency caused by random reads when accessing memory regions that exceed the size of the last level cache [20, 31]. That is possible because switching between coroutines takes less time than reading from memory. Additionally, the authors praise the simplicity of converting existing synchronous functions into coroutines and experimentally confirm that the performance of C++-Coroutines is as good as the performance of other approaches for instruction stream interleaving.

In [32], Psaropoulos et al. explore the usage of coroutines to hide the higher-than-DRAM latency of accessing non-volatile memory. They conclude that interleaving with coroutines narrows the performance gap between NVM and DRAM for latency-bound operations as long as there is enough independent work to execute in the meantime.

He et al. propose a coroutine-to-transaction model in [18]. In this model, they start one coroutine for each transaction. Upon a possible CPU-cache miss, they switch the execution from one transaction to another. As long as there are multiple concurrent transactions, they can find enough independent work to hide a CPU-cache miss. This design allows them to avoid code changes as much as possible while getting the benefits of memory prefetching. Although not discussed in their paper, this design should be easily adaptable to asynchronously read from storage.

7 SUMMARY AND FUTURE WORK

In the last ten years, SSDs made astonishing improvements in capacity per dollar and performance. We are convinced that they are now the ideal storage technology for scan-oriented OLAP database systems. However, we need a lot of parallel I/O requests to exploit them, and the latency of reading from them is still very high.

In this paper, we proposed to use asynchronous I/O and coroutines to generate a lot of parallel I/O requests and hide the I/O latency. First, we showed that this enables us to get significantly higher throughput with less compute resources. Additionally, we found that this substantially flattens the performance cliff when exceeding main memory. Then, we discussed how to integrate support for coroutines into a code-generating DBMS and showed how we can call C++-Coroutines from Codegen-Coroutines. Finally, we presented our implementation of coroutines for asynchronous I/O in Umbra and measured performance improvements of up to 60% for our new asynchronous index-nested-loop joins.

For future work, we plan to significantly extend Umbra's support of asynchronous I/O. As previously mentioned, we plan to add support for asynchronous locks. Furthermore, we want to make our task-scheduler coroutine aware to implement a coroutine-per-morsel approach. Finally, we want to add support for asynchronous I/O to all our scan operators.

REFERENCES

[1] [n.d.]. *Coroutines.* Retrieved October 25, 2021 from https://en.cppreference.com/w/cpp/language/coroutines

[2] [n.d.]. *Samsung 980 PRO PCIe 4.0 NVMe SSD.* Retrieved March 27, 2022 from https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-w-heatsink-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0cw/

[3] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2205–2217. https://doi.org/10.1145/3514221.3526045

[4] Jens Axboe. 2019. *Efficient IO with io_uring.* Retrieved October 25, 2021 from https://kernel.dk/io_uring.pdf

[5] Lewis Baker. [n.d.]. *Asymmetric Transfer.* Retrieved October 25, 2021 from https://lewissbaker.github.io

[6] Lewis Baker. [n.d.]. *CppCoro - A coroutine library for C++.* https://github.com/lewissbaker/cppcoro

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[8] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, Danica Porobic and Thomas Neumann (Eds.). ACM, 2:1–2:8. https://doi.org/10.1145/3399666.3399908

[9] ClickHouse, Inc. 2022. ClickHouse: Fast Open-Source OLAP DBMS. https://clickhouse.com/.

[10] Transaction Processing Performance Council. 2021. *TPC Benchmark H.* Retrieved October 25, 2021 from http://tpc.org/tpc_documents_current_versions/pdf/tpch_v3.0.0.pdf

[11] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR 2022, Conference on Innovative Data Systems Research.* https://db.cs.cmu.edu/papers/2022/p13-crotty.pdf

[12] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[13] Franz Faerber, Alfons Kemper, Per Åke Larson, Justin Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130. https://doi.org/10.1561/1900000058

[14] Emmanuel Goossaert. 2014. *Coding for SSDs.* Retrieved October 25, 2021 from https://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/

[15] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170. https://doi.org/10.1145/152610.152611

[16] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf

[17] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981992. https://doi.org/10.1145/1376616.1376713

[18] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 431444.

[19] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends Databases* 1, 2 (Feb. 2007), 141259. https://doi.org/10.1561/1900000002

[20] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (July 2018), 17021714. https://doi.org/10.14778/3236187.3236216

[21] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP amp;OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[22] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905. https://doi.org/10.1007/s00778-020-00643-4

[23] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (dec 2015), 252263. https://doi.org/10.14778/2856318.2856321

[24] Viktor Leis. 2021. *What Every Programmer Should Know About SSDs.* Retrieved October 25, 2021 from https://databasearchitects.blogspot.com/2021/06/what-every-programmer-should-know-about.html

[25] 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 *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[26] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. 185–196. https://doi.org/10.1109/ICDE.2018.00026

[27] David Lomet. 2018. Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) *(DAMON '18)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/3211922.3211927

[28] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

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

[30] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 2:1–2:11. https://doi.org/10.1145/3465998.3466003

[31] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 230242. https://doi.org/10.14778/3149193.3149202

[32] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-Bound Operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3329785.3329917

[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212