Umbra: A Disk-Based System with In-Memory Performance

Thomas Neumann, Michael Freitag
Technische Universität München
{neumann,freitagm}@in.tum.de

ABSTRACT

The increases in main-memory sizes over the last decade have made pure in-memory database systems feasible, and in-memory systems offer unprecedented performance. However, DRAM is still relatively expensive, and the growth of main-memory sizes has slowed down. In contrast, the prices for SSDs have fallen substantially in the last years, and their read bandwidth has increased to gigabytes per second. This makes it attractive to combine a large in-memory buffer with fast SSDs as storage devices, combining the excellent performance for the in-memory working set with the scalability of a disk-based system.

In this paper we present the Umbra system, an evolution of the pure in-memory HyPer system towards a disk-based, or rather SSD-based, system. We show that by introducing a novel low-overhead buffer manager with variable-size pages we can achieve comparable performance to an in-memory database system for the cached working set, while handling accesses to uncached data gracefully. We discuss the changes and techniques that were necessary to handle the out-of-memory case gracefully and with low overhead, offering insights into the design of a memory optimized disk-based system.

1. INTRODUCTION

Hardware trends have greatly affected the development and evolution of database management systems over time. Historically, most of the data was stored on (rotating) disks, and only small fractions of the data could be kept in RAM in a buffer pool. As main memory sizes grew significantly, up to terabytes of RAM, this perspective changed as large fractions of the data or even all data could now be kept in memory. In comparison to disk-based systems, this offered a huge performance advantage and led to the development of pure in-memory database systems [4, 5], including our own system HyPer [9]. These systems make use of RAM-only storage and offer outstanding performance, but tend to fail or degrade heavily if the data does not fit into memory.

Moreover, we currently observe two hardware trends that cast strong doubt on the viability of pure in-memory systems. First, RAM sizes are not increasing significantly any more. Ten years ago, one could conceivably buy a commodity server with 1 TB of memory for a reasonable price. Today, affordable main memory sizes might have increased to 2 TB, but going beyond that disproportionately increases the costs. As costs usually have to be kept under control though, this has caused the growth of main memory sizes in servers to subside in the recent years.

On the other hand, SSDs have achieved astonishing improvements over the past years. A modern 2 TB M.2 SSD can read with about 3.5 GB/s, while costing only $500. In comparison, 2 TB of server DRAM costs about $20 000, i.e. a factor of 40 more. By placing multiple SSDs into one machine we can get excellent read bandwidths at a fraction of the cost of a pure DRAM solution. Because of this, Lomet argues that pure in-memory DBMSs are uneconomical [15]. They offer the best possible performance, of course, but they do not scale beyond a certain size and are far too expensive for most use cases. Combining large main memory buffers with fast SSDs, in contrast, is an attractive alternative as the cost is much lower and performance can be nearly as good.

We wholeheartedly agree with this notion, and present our novel Umbra system which simultaneously features the best of both worlds: Genuine in-memory performance on the cached working set, and transparent scaling beyond main memory where required. Umbra is the spiritual successor of our pure in-memory system HyPer, and completely eliminates the restrictions of HyPer on data sizes. As we will show in this paper, we achieve this without sacrificing any performance in the process. Umbra is a fully functional general-purpose DBMS that is actively developed further by our group. All techniques presented in this paper have been implemented and evaluated within this working system. While Umbra and HyPer share several design choices like a compiling query execution engine, Umbra deviates in many important aspects due to the necessities of external memory usage. In the following, we present key components of the system and highlight the changes that were necessary to support arbitrary data sizes without losing performance for the common case that the entire working set fits into main memory.

A key ingredient for achieving this is a novel buffer manager that combines low-overhead buffering with *variable-size pages*. Compared to a traditional disk-based system, in-memory systems have the major advantage that they can do away with buffering, which both eliminates overhead and greatly simplifies the code. For disk-based systems, common wisdom dictates to use a buffer manager with fixed-size pages. However, while this simplifies the buffer manager itself, it makes using the buffer manager exceedingly difficult. For example, large strings or lookup tables for dictionary compression often cannot easily be stored in a single fixed-size page, and both complex and expensive mechanisms are thus required all over the database system in order to handle large objects. We ar-

10th Annual Conference on Innovative Data Systems Research (CIDR ‘20). January 12-15, 2020, Amsterdam, Netherlands.

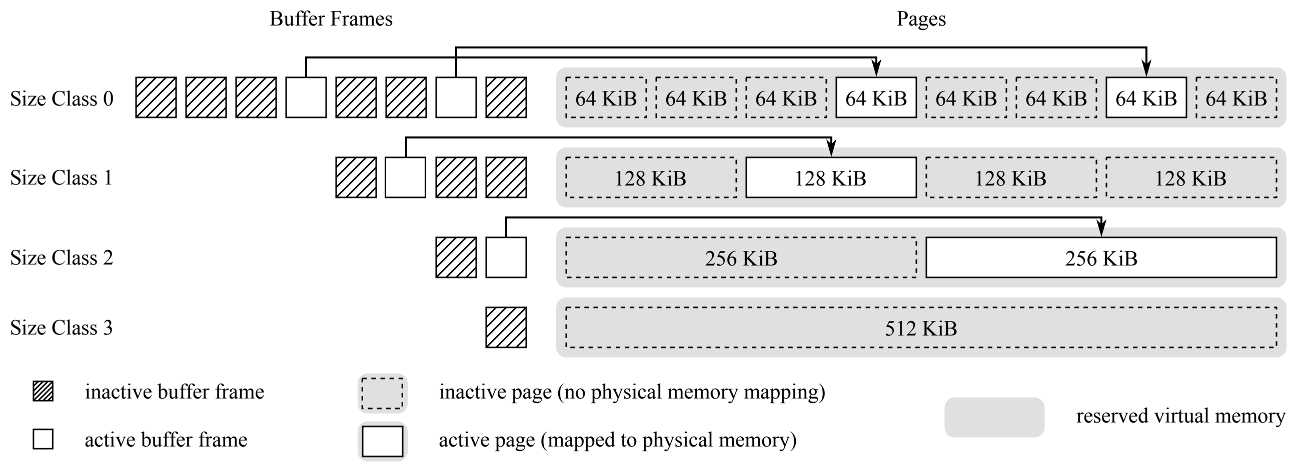

Figure 1: Illustration of the buffer manager, assuming a buffer pool size of 512 KiB and a minimum page size of 64 KiB. The buffer manager supports exponentially growing page sizes which are organized into size classes. For each size class, a virtual memory region the size of the entire buffer pool is reserved, and buffer frames correspond to fixed addresses within this memory region.

gue that it is much better to use a buffer manager with variable-size pages, which allows for storing large objects natively and consecutively if needed. Such a design leads to a more complex buffer manager, but it greatly simplifies the rest of the system. If we can rely upon the fact that a dictionary is stored consecutively in memory, decompression is just as simple and fast as in an in-memory system. In contrast, a system with fixed-size pages either needs to re-assemble (and thus copy) the dictionary in memory, or has to use a complex and expensive lookup logic.

Of course, there are substantial technical reasons why previous systems have preferred fixed-size pages, such as fragmentation issues. However, we show in this work how these problems can be eliminated by exploiting the dynamic mapping between virtual addresses and physical memory provided by the operating system. In summary, we make the following contributions. First, we present a novel buffer manager that supports variable-size pages and introduces only minimal overhead (Section 2). Second, we highlight further key adjustments that are necessary to transition to a disk-based system (Section 3). In particular, we provide insights into string handling, statistics maintenance, and the execution model in Umbra. Finally, we present experiments in Section 4, review related work in Section 5, and draw conclusions in Section 6.

2. BUFFER MANAGER

Previous research has shown that traditional buffer managers are one of the major bottlenecks when deploying database management systems on modern hardware platforms [7]. For this reason, we recently published the LeanStore storage manager that overcomes these inefficiencies [14]. LeanStore is a highly scalable buffer manager that offers nearly the same performance as a pure in-memory system when the working set fits into main memory. However, it still relies on fixed-size pages and thus, as outlined above, requires expensive mechanisms to handle large objects. In Umbra, we go one step further and build upon the fundamental ideas proposed by LeanStore, while additionally supporting variable-size pages.

Database pages in Umbra are conceptually organized in *size classes*, where a size class contains all pages of a given size. Size class 0 contains the smallest pages, which should be a multiple of the system page size. In Umbra, we choose 64 KiB as the smallest page size. Subsequent size classes contain pages of exponentially growing size, i.e. pages in size class $i+1$ are twice as large as those in size class i (cf. Figure 1). Pages can theoretically be as large as the entire buffer pool, although in practice even the largest pages are much smaller than this theoretical limit.

Our buffer manager maintains a *single* buffer pool with a configurable size, into which pages from *any* size class can be loaded. By default, we allow the buffer pool to occupy half of the available physical memory, leaving the other half as scratch memory for query execution. Crucially, Umbra does not require that the amount of buffer pool memory is configured individually per page size class as it is necessary in previous systems that support variable-size pages [18]. For this reason, the external interface of the proposed buffer manager does not differ significantly from a traditional buffer manager. That is, the buffer manager exposes functions which cause a specific page to be pinned in memory, loading it from disk if required, and functions that cause a page to be unpinned, allowing it to be subsequently evicted from memory.

2.1 Buffer Pool Memory Management

The major challenge in implementing a buffer manager that supports multiple page sizes within a single buffer pool is external fragmentation in this buffer pool. Fortunately, we can avoid this problem by exploiting the flexible mapping between virtual addresses and physical memory provided by the operating system. The operating system kernel maintains a page table to transparently translate the virtual addresses that are used by user-space processes to physical addresses within the actual RAM. This not only allows contiguous blocks of virtual memory to be physically fragmented, but also enables virtual memory to be allocated independently of physical memory. That is, an application can reserve a block of virtual memory for which the kernel does not immediately create a mapping to physical memory within the page table.

These particular properties of virtual memory management are exploited within our buffer manager to completely avoid any external fragmentation within the buffer pool. In particular, the buffer manager uses the `mmap` system call to allocate a separate block of virtual memory for each page size class, where each one of these memory regions is large enough to theoretically accommodate the entire buffer pool. We configure the `mmap` call to create a private anonymous mapping which causes it to simply reserve a contiguous range of virtual addresses which do not yet consume any physical memory (cf. Figure 1). Subsequently, each of these virtual memory regions is partitioned into page-sized chunks, and one buffer frame containing a pointer to the respective virtual address is created for each chunk. These pointers identify the virtual addresses at which page data can be stored in memory and remain static for the entire lifetime of the buffer manager. Since page sizes are fixed within a given size class and a separate virtual address range is reserved for each size class, no fragmentation of the *virtual* address space associated with a size class occurs. Of course, the physical memory that is used to store the page data associated with an active buffer frame may still be fragmented.

When a buffer frame becomes active, the buffer manager uses the `pread` system call to read data from disk into memory. This data is stored at the virtual memory address associated with the buffer frame, at which point the operating system creates an actual mapping from these virtual addresses to physical memory (cf. Figure 1). If a previously active buffer frame becomes inactive due to eviction from the buffer pool, we first write any changes to the page data back to disk using the `pwrite` system call, and subsequently allow the kernel to immediately reuse the associated physical memory. On Linux, this can be achieved by passing the `MADV_DONTNEED` flag to the the `madvise` system call. This step is critical to ensure that the physical memory consumption of the buffer pool does not exceed the configured buffer pool size, as several times more virtual memory is allocated internally (cf. Figure 1). As the memory mappings used in the buffer manager are not backed by any actual files (see above), the `madvise` call incurs virtually no overhead.

We currently assume an underlying block device storage abstraction, which allows us to rely on the `pread` and `pwrite` system calls to move data between background storage and main memory. This both simplifies the implementation and improves the flexibility of our buffer manager since it imposes no constraints on the actual hardware that is used for background storage. Nevertheless, direct communication with this hardware without an intermediate block device abstraction, e.g. through open-channel SSDs, would provide various interesting optimization opportunities which we plan to explore in the future [3].

During operation, the buffer manager keeps track of the total size of all pages that are currently held in memory. It ensures that their total size never exceeds the maximum size of the buffer pool by evicting pages to disk as required. Umbra employs essentially the same replacement strategy as LeanStore [14], where we speculatively unpin pages but do not immediately evict them from main memory. These cooling pages are then placed in a FIFO queue, and eventually evicted if they reach the end of the queue.

Overall, this approach allows the Umbra system to fully utilize the benefits of variable-size pages, with minimal runtime overhead and implementation complexity. However, variable-size pages alone do not resolve all the shortcomings of a traditional buffer manager in a modern database system [14]. In the following, we give a brief overview of the additional optimizations found in LeanStore that we adapt for variable-size pages.

2.2 Pointer Swizzling

Since pages are serialized to disk, they need to be referenced through logical page identifiers (PIDs) in the general case. However, centralized approaches which rely on a global hash table to map PIDs to memory addresses in the buffer manager can quickly become a major performance bottleneck in modern many-core systems [7]. In Umbra, we instead rely on pointer swizzling as a low-overhead decentralized technique for address translation [14].

In this approach, references to both memory-resident and disk-resident pages are implemented through *swips*, which encode all information that is required to locate and access pages. A swip is a single 64-bit integer which contains either a virtual memory address, in case the referenced page resides in memory, or a 64-bit PID if it currently resides on disk. A swip is said to be *swizzled* if it references a memory-resident page, and *unswizzled* otherwise. We use pointer tagging to distinguish between these two options, and thus only a single additional conditional statement is required to access a memory-resident page. In addition to the tagging bit, an unswizzled swip stores both the size class of the corresponding page (6 bits), and its actual page number (57 bits). This way, the buffer manager requires no additional information besides a swip to load the corresponding page into memory (cf. Figure 2).

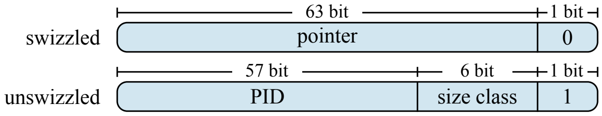

Figure 2: Illustration of a swizzled (top) and unswizzled (bottom) swip. A swizzled swip stores a pointer to a memory-resident page, the lowest bit of which will be zero due to the mandatory 8-byte alignment of pointers. In an unswizzled swip this bit is fixed to one, while the remaining bits store the page identifier and the size class of a page residing on disk.

Due to the decentralized nature of pointer swizzling, some operations such as page eviction become more complicated. For example, the same page could be referenced by several swips, all of which would need to be updated when the page is evicted to disk. However, we cannot easily locate all swips that reference a given page, which makes it hard to maintain consistency [14]. To counter this, all buffer-managed data structures in Umbra are required to organize their constituent pages in a (possibly degenerate) tree. This ensures by design that each page is referenced by exactly one owning swip, and no further swips besides that owning swip need to be updated when a page is loaded or evicted. In case of B$^+$-trees, for instance, this entails that leaf pages may not contain references to their siblings, as this would require that leaf pages are referenced by more than one swip. We will outline below how efficient scans can still be implemented in light of these restrictions.

2.3 Versioned Latches

Frequent latch acquisitions for thread synchronization within the buffer manager quickly become another point of contention on modern hardware platforms [14]. For this reason LeanStore uses optimistic latching to synchronize concurrent accesses to the same page, allowing buffer-managed data structures to drastically reduce the number of actual latch acquisitions. Note that thread synchronization is independent of transaction concurrency control which has to be implemented on top of these data structures.

We adopt the optimistic latching scheme proposed by LeanStore within Umbra, which is implemented as follows. Each active buffer frame contains a *versioned latch* which can either be acquired in *exclusive* mode, *shared* mode, or *optimistic* mode. The main component of a versioned latch is a single 64-bit integer, of which 5 bits are used to store state information, while the remaining 59 bits are used to store a version counter. The state bits encode if and in which mode a latch is currently locked, and the version counter is used to validate optimistic accesses to a buffer frame (cf. Figure 3). Versioned latches are accessed and modified through atomic operations to ensure proper synchronization.

A latch is unlocked if its state bits are set to the integer value zero. An unlocked latch can be acquired in exclusive mode by atomically setting its state bits to the integer value 1. At most one thread at a time is allowed to hold a latch in exclusive mode, which then acts similar to a global mutex. For example, any modifica-

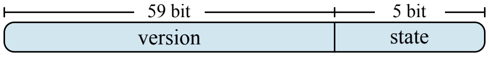

Figure 3: Structure of the versioned latch stored in a buffer frame for synchronization of page accesses. The version counter is increased after each modification of a page, allowing optimistic accesses to be validated. The state bits encode whether and in which mode the latch is locked.

tion of a page, such as inserting data, requires that the corresponding latch is first acquired in exclusive mode in order to avoid data races. A thread releases an exclusive latch by resetting the state bits to zero. Additionally, the version counter is incremented in case the page data was modified in any way.

Alternatively, multiple threads can acquire a latch simultaneously in shared mode, provided that it is not currently locked in exclusive mode by another thread. A shared latch has its state bits set to an integer value greater than 1, where a value of $n+1$ indicates that n threads currently hold the latch in shared mode. In the rare case that the state bits are not sufficient to count the number of threads, an additional 64-bit integer is used as an overflow counter. No modifications of a page are allowed while holding a shared latch, but read operations are guaranteed to succeed. A shared latch effectively pins the associated page in the buffer manager, preventing it from being unloaded. If other threads still hold a shared latch, it is released by simply decrementing the thread count encoded in the state bits. The last thread that releases a shared latch fully unlocks it by resetting the state bits to zero.

Finally, a latch that is unlocked or locked in shared mode can be acquired by any number of threads in optimistic mode. This is achieved by simply remembering the value of the version counter at the time of latch acquisition, i.e. no modification of the latch itself is required and thus no contention is induced. Like in shared mode, only read accesses to a page are allowed while holding an optimistic latch. However, these read accesses are allowed to fail, since another thread could acquire an exclusive latch and modify the page concurrently. Therefore, all optimistic accesses have to be validated when an optimistic latch is released. If the version counter changed since the acquisition of the latch, a concurrent modification of the page occurred and the read operations are restarted. Note that it is even legal for a page to be unloaded while the page content is being read optimistically. This is possible since the virtual memory region reserved for a buffer frame always remains valid (see above), and read accesses to a memory region that was marked with the `MADV_DONTNEED` flag simply result in zero bytes. No additional physical memory is allocated in this case, as all such accesses are mapped to the same zero page. Optimistic latching eliminates contention on the latches themselves if there are many concurrent read accesses [14].

While LeanStore only supports optimistic latching for readers, we additionally support shared latching [14]. This is required as operators in Umbra are generally oblivious of the paged nature of relations. If we only supported optimistic latching for readers, every operator in a pipeline would have to include additional validation logic in case a page was evicted while being processed. Furthermore, this avoids frequent validation failures in read-heavy OLAP queries in case other queries write to the same relation.

2.4 Buffer-Managed Relations

Building on the basic facilities provided by the buffer manager, relations in Umbra are organized in B$^+$-trees, using synthetic 8-byte tuple identifiers as keys. The identifier for a given tuple is generated at the time the tuple is inserted into the B$^+$-tree. We ensure that tuple identifiers increase strictly monotonically, which allows us to avoid splitting nodes during tuple insertions. Instead, we completely fill existing inner and leaf nodes before allocating new nodes. Using synthetic keys in the primary B$^+$-trees additionally offers the major advantage that arbitrary insert patterns are handled equally well by Umbra. Inner nodes always use the smallest available page size (64 KiB), leading to a fanout of 8 192. Since we never split leaf nodes, they are only allocated when a tuple does not fit into an existing page during an insert operation. In this case, the smallest page size that can fit the entire tuple is chosen. Usually, this single tuple fits easily on a 64 KiB page and as a result the majority of leaf pages in Umbra will also be 64 KiB large. In practice, we found that this provides a good balance between efficient point accesses and efficient range accesses, e.g. for scans.

Tuples within a leaf page are currently organized in a PAX layout [1]. That is, the fixed-size attributes are stored in a columnar layout at the start of the page, and the variable-size attributes are stored densely at the end of the page. Pages are compactified during inserts if required. However, such a page layout is not optimal for data that resides on disk, as all attributes of a relation have to be loaded even if only few are actually accessed. Thus, we also plan to integrate alternative storage layouts in the future, such as DataBlocks for cold parts of the data [11].

Concurrent access to the B$^+$-trees is synchronized through optimistic latch coupling which makes heavy use of the versioned latches associated with each buffer frame [14]. Writers acquire latches in exclusive mode only to split inner pages or insert tuples into leaf pages, while readers only acquire latches in shared mode in order to read tuples from a leaf page or to load a child page from disk. During non-modifying traversal, latches are acquired in optimistic mode, and validated where required. Due to the restriction that each page has exactly one owning swip, there are no links between adjacent leaf nodes like in a traditional B$^+$-tree. In order to avoid frequent full traversals of the tree, we maintain an optimistic latch on the parent of the current leaf node during table scans. As long as this optimistic latch remains valid, i.e. no concurrent modifications of the parent node occur, this allows us to cheaply navigate to the next leaf node.

2.5 Recovery

The Umbra system uses ARIES for recovery [16]. Overall, ARIES seamlessly supports the varying page sizes employed by Umbra. However, some care has to be taken in order to ensure recoverability when reusing disk space. In particular, we cannot store multiple smaller pages in disk space that was previously occupied by a single large page. Consider, for example, a 128 KiB database file which is currently entirely occupied by a single 128 KiB page. We now load this page into memory, delete it, and create two new 64 KiB pages that reuse the disk space in the database file. If the system crashes, it is possible that we only manage to write the corresponding log records to disk, but not the actual new page data. During recovery, ARIES would then at some point attempt to read the log sequence number of the second 64 KiB page from the database file, although it has never been written to disk. Thus, it would actually read some data of the deleted 128 KiB page and incorrectly interpret it as a log sequence number. In order to avoid such problems, Umbra only reuses disk space for pages of the same size.

3. FURTHER CONSIDERATIONS

While its buffer manager is certainly the key ingredient that enables in-memory performance in Umbra, further components had

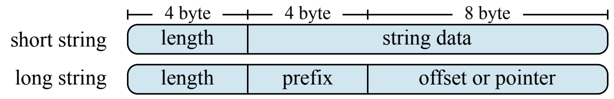

Figure 4: Structure of the 16-byte string headers in Umbra.

to be adjusted or redesigned in comparison to HyPer. The most significant adjustments in Umbra were made to string handling and statistics collection, as well as compilation and execution.

3.1 String Handling

Umbra stores string attributes in two separate parts, a 16-byte header containing metadata, and a variable-size body containing the actual string data. The header is stored like any other fixed-size attribute in the columnar layout at the start of a page, while the variable-size actual string data is stored at the end of a page. Since our buffer manager supports multiple page sizes, we do not have to split long strings across several pages.

Depending on the string length, the string header representation in Umbra will differ slightly (cf. Figure 4). The first four bytes of the header always contain the length of the string, i.e. string length is limited to $2^{32} - 1$ in Umbra. Short strings that contain 12 or fewer characters are stored directly within the remaining 12 bytes of the string header, thus avoiding an expensive pointer indirection. Longer strings are stored out-of-line, and the header will contain either a pointer to their storage location, or an offset from a known location. Generally, strings that are stored in database pages are addressed by offsets from the page start, and other strings are addressed by pointer. In case of long strings, the remaining four bytes of the header are used to store the first four characters of the string, allowing Umbra to short-circuit some comparisons.

As opposed to a pure in-memory system, a disk-based system like Umbra cannot guarantee that pages are retained in memory during the entire query execution time. Therefore, strings that are stored out-of-line require some special care, as the offsets or pointers stored in their header may become invalid if the corresponding page is evicted. For this purpose, Umbra introduces three storage classes for out-of-line strings, namely *persistent*, *transient*, and *temporary* storage. The storage class is encoded within two bits of the offset or pointer value stored in the string header.

References to a string with persistent storage, e.g. query constants, remain valid during the entire uptime of the database. References to a string with transient storage duration are valid while the current unit of work is being processed, but will eventually become invalid. Unlike persistent strings, transient strings need to be copied if they are materialized during query execution. Any string that originates from a relation, for example, has transient storage in Umbra, as the corresponding page could be evicted from memory. Finally, strings that are actually *created* during query execution, e.g. by the `UPPER` function, have temporary storage duration. While temporary strings can be kept alive as long as required, they have to be garbage collected once their lifetime ends.

3.2 Statistics

Umbra tracks a number of statistics for query optimization purposes. First of all, a random sample of each relation is maintained. Since Umbra is designed as a disk-based system, straightforward random sampling from the base relations is not feasible. In contrast to a pure in-memory system, computing the sample on-demand is prohibitively expensive in a disk-based system. Even in an in-memory system, computing a random sample remains an expensive operation. HyPer alleviates this problem by only periodically updating the sample, but then samples become outdated very quickly in write-heavy OLTP workloads. In Umbra, we instead implement a scalable online reservoir sampling algorithm that we recently developed [2]. In doing so, we can ensure that the query optimizer always has access to an up-to-date sample with minimal overhead.

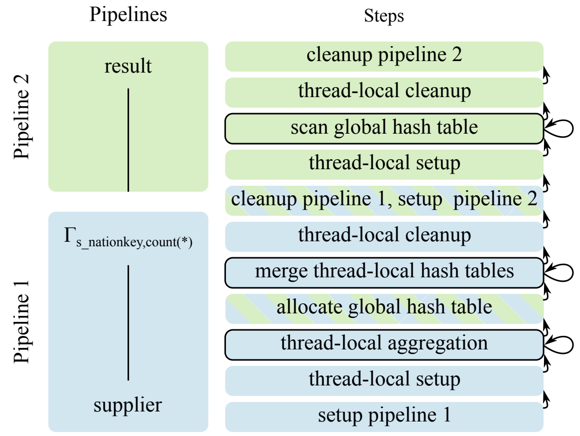

Figure 5: Pipelines and the corresponding steps for a simple group-by query in Umbra. Parallel steps are marked by a black border, and arrows indicate state transitions between the steps.

Besides a random sample, we additionally maintain updateable HyperLogLog sketches on each individual column. As we have shown previously [6], our implementation of updateable HyperLogLog sketches can provide almost perfect cardinality estimates on individual columns with moderate overhead. Furthermore, this enables the query optimizer to utilize highly accurate multi-column estimates through a combination of sketch-based and sampling-based estimation [6].

3.3 Compilation & Execution

In general, Umbra employs the same query execution strategy as HyPer: Logical query plans are translated into efficient parallel machine code, which is then executed to obtain the query result [17]. While they naturally exhibit various similarities, Umbra develops the HyPer approach further in several key ways.

First of all, Umbra employs a much more fine-grained representation of physical execution plans than HyPer. In HyPer, a physical execution plan is essentially a monolithic code fragment which is compiled as a whole and computes the query result [17]. In contrast, physical execution plans in Umbra are represented as modular state machines. Consider, for example, the following query on the well-known TPCH schema:

```
select   count(*)
from     supplier
group by s_nationkey
```

On a high-level, the execution plan of this query consists of two pipelines, where the first pipeline scans the `supplier` table and performs the `group by` operation, and the second pipeline scans the groups and prints the query output (cf. Figure 5). In Umbra, these pipelines are further disassembled into *steps*, which can either be single-threaded or multi-threaded. In particular, Umbra would generate the steps shown in Figure 5 for the above query.

In generated code, each step corresponds to a separate function which can be called by the runtime system of Umbra. For the purpose of query execution, these individual steps are viewed as states with well-defined transitions between steps, which are orchestrated by the query executor of Umbra (cf. Figure 5). In case of multi-threaded steps, a morsel-driven approach is employed to distribute the available work to worker threads, and the step function processes a single morsel on each invocation [12].

The modular execution plan model employed by Umbra offers several crucial benefits. First, we can suspend query execution after each invocation of a step function, e.g. if the system IO load exceeds some threshold. Furthermore, our query executor can detect at runtime whether a parallel step would only consist of a single morsel. In this case, the required work does not have to be dispatched to another thread, avoiding a potentially expensive context switch. Finally, we can easily support multiple parallel steps within a pipeline, as illustrated in Figure 5.

Another important difference is that query code is not generated directly in the LLVM intermediate representation (IR) language. Instead, we implement a custom lightweight IR in Umbra, which allows us to generate code efficiently without relying on LLVM. Since LLVM is designed as a versatile general-purpose code generation framework, it can incur a noticeable overhead due to functionality that is not required by Umbra anyway. By implementing a custom IR, we can avoid this potentially expensive roundtrip through LLVM during code generation.

Unlike HyPer, Umbra does not immediately compile this IR to optimized machine code. Instead, we employ an adaptive compilation strategy which strives to optimize the tradeoff between compilation and execution time for each individual step [10]. Initially, the IR associated with a step is always translated into an efficient bytecode format and interpreted by a virtual machine backend. For parallel steps, the adaptive execution engine then tracks progress information to decide whether compilation could be beneficial [10]. If applicable, the Umbra IR is translated into LLVM IR and compilation is delegated to the LLVM just-in-time compiler. Conceptually, our IR language is designed to closely resemble a subset of the LLVM IR, such that translation from our format into LLVM format can be achieved cheaply and in linear time.

4. EXPERIMENTS

Our experimental evaluation consists of two parts. First, we compare Umbra to its spiritual predecessor HyPer (v0.6-165) and the well-known column store MonetDB (v11.33.3), in order to demonstrate its competitive performance [8, 9]. Second, we highlight key characteristics of Umbra in additional microbenchmarks. All benchmarks are performed on an Intel Core i7-7820X CPU with 8 physical and 16 logical cores running at 3.6 GHz. The system contains 64 GiB of main memory, and all database files are placed on a Samsung 960 EVO SSD with 500 GiB of storage space.

4.1 Systems Comparison

As the basis for our comparative experiments, we choose the join order benchmark JOB [13] and the TPCH benchmark at scale factor 10. Each query is repeated five times, and we report the fastest repetition (i.e. we measure performance with warm caches in this experiment). The relative performance of Umbra in comparison to HyPer and MonetDB is shown in Figure 6.

We observe that Umbra performs excellently in comparison to HyPer. In particular, Umbra achieves a geometric mean speedup of 3.0× on JOB and of 1.8× on TPCH. To some extent, this notably large gap can be attributed to the much more efficient adaptive compilation strategy employed by Umbra. Whereas HyPer always compiles query plans to optimized machine code, the adaptive strategy avoids expensive compilation for cheap queries. On such queries, HyPer actually spends far more time on query compilation than on query execution, by up to 29×.

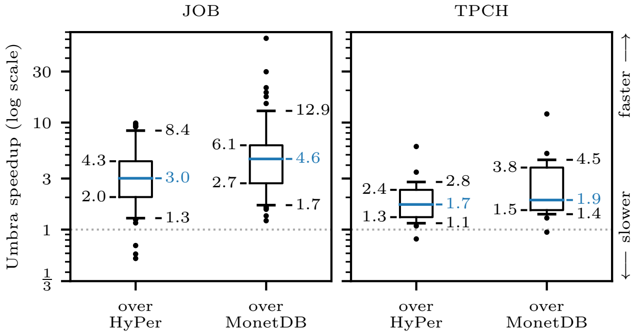

Figure 6: Relative speedup of Umbra over HyPer and MonetDB on JOB and TPCH. The boxplots show the 25th and 75th percentiles (left), and the 5th, 50th, and 95th percentiles (right).

More importantly, if we look at raw query execution time alone, Umbra achieves comparable performance to the pure in-memory system HyPer. On average, execution times fluctuate by 30 % on JOB and 10 % on TPCH. While larger relative differences between 0.2× and 2.3× on JOB and 0.5× and 1.6× on TPCH do occur, we determined that these outliers, especially on JOB, mostly arise due to different logical plans chosen by HyPer and Umbra. In fact, we found that on some queries Umbra happens to pick a worse logical plan than HyPer despite having much better cardinality estimates, because the query optimizer in HyPer was in luck and made two mistakes that canceled out.

The geometric mean speedup of Umbra over MonetDB is 4.6× on JOB and 2.3× on TPCH. On many queries, in particular the more complex JOB queries, MonetDB spends a large fraction of the overall query runtime on query optimization and code generation. Furthermore, unlike HyPer, MonetDB also occasionally picks extremely bad query plans which lead to slow query execution times. Even with good plans, however, query execution in MonetDB is generally less efficient than in HyPer and Umbra.

4.2 Microbenchmarks

In the following, we will investigate further which impact some key components of Umbra have on its performance. We begin by demonstrating that Umbra incurs only minimal overhead due to its buffer manager. For this, we modify the storage layer in two different ways. First, we disable page eviction in the buffer manager which eliminates the necessity to materialize strings originating from database pages into memory. Second, we implement an alternative storage layer which simply stores relations in flat memory-mapped files without any buffer manager. This additionally eliminates the overhead of the buffer manager itself, and the indirections incurred by the B^+-tree representation of relations.

We then run JOB and TPCH on these modified storage layers and investigate the changes in query execution time. Optimization and compilation time are disregarded in this case, as Umbra uses the exact same logical and physical plans with all three storage layers. We observe that performance fluctuates only slightly in both directions, by less than 2 % on average if only string materializations are avoided, and by less than 6 % on average if the entire buffer manager is bypassed. In both cases, the most extreme changes in performance still amount to just 30 %. From these results, we conclude that the buffer manager incurs negligible costs in Umbra.

We utilize the modified storage layer for another microbenchmark illustrating the I/O performance of the buffer manager. For this, we generate a relation containing 250 million random 8-byte floating point values. Subsequently, we compute the sum of these values with cold database and OS caches. When bypassing the buffer manager, Umbra relies on the operating system to load the contents of the flat memory-mapped file containing the relation data into memory. In this case, Umbra achieves a read throughput of 1.15 GiB/s which we determined to be close to the maximum available random access bandwidth on our SSD. Read accesses to the SSD happen essentially at random due to the highly parallelized nature of query execution in Umbra which imposes no inter-thread constraints on the order in which pages are processed. When actually utilizing the buffer manager, Umbra achieves virtually the same read throughput of 1.13 GiB/s. These results show that the buffer manager in Umbra can make full use of the available I/O bandwidth. We further verified this finding by running JOB and TPCH with artificially reduced buffer pool sizes, in which case Umbra becomes similarly I/O-bound.

In summary, we observe that the primary bottleneck in Umbra is caused by limited storage throughput, whereas the buffer manager itself is easily capable of handling highly parallel and I/O-heavy workloads. As suggested in Section 1, we can increase the available I/O bandwidth by simply placing multiple SSDs into a single system, allowing Umbra to rival the performance of a pure in-memory system even on workloads that do not fit into main memory alone.

5. RELATED WORK

As a re-design of our HyPer system, Umbra builds heavily on the insights gained during the development of HyPer [9]. We concur with the argument set forth by Lomet that pure in-memory systems are uneconomical [15], and present Umbra as an evolution of HyPer towards a high-performance disk-based system. As such, the system requires an efficient and scalable buffer manager, which requires careful adaptations to the architecture of a traditional buffer manager. The buffer manager in Umbra shares many properties with LeanStore [14], and we refer the reader to [14] for a detailed review of the relevant literature. We furthermore argue that it is desirable to support variable-size pages in the buffer manager to reduce the complexity of handling large data objects. To the best of our knowledge, the only other buffer manager with variable-size pages was developed for the ADABAS system [18]. However, in comparison to Umbra, this buffer manager is much less flexible as it only supports two different page sizes that are maintained in separate pools. A major downside of their approach is that we have to specify in advance how much memory is allocated for each pool.

6. CONCLUSION

In this paper we presented the newly developed Umbra system that constitutes an evolution of the purely in-memory system HyPer towards an SSD-based system. We demonstrated that Umbra can achieve in-memory performance if the entire working set fits into RAM, while at the same time fully utilizing the available I/O bandwidth if data has to be spilled to disk. We introduced a novel, low-overhead buffer manager with variable-size pages that renders this kind of performance possible. Furthermore, we investigated key adaptations to other components of an in-memory database system that are required for the transition to an SSD-based system. Our findings can serve as a guideline for the design of a novel breed of high-performance data caching systems.

7. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[2] A. Birler. Scalable reservoir sampling on many-core CPUs. In *SIGMOD*, pages 1817–1819, 2019.

[3] M. Bjørling, J. Gonzalez, and P. Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *FAST*, pages 359–374. USENIX Association, 2017.

[4] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[5] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[6] M. Freitag and T. Neumann. Every row counts: Combining sketches and sampling for accurate group-by result estimates. In *CIDR*, 2019.

[7] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[8] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[9] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[10] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. In *ICDE*, pages 197–208, 2018.

[11] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, pages 311–326, 2016.

[12] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[13] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[14] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-memory data management beyond main memory. In *ICDE*, pages 185–196, 2018.

[15] D. B. Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *DaMoN*, pages 9:1–9:10, 2018.

[16] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[17] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[18] H. Schöning. The ADABAS buffer pool manager. In *VLDB*, pages 675–679, 1998.