Robust External Hash Aggregation in the Solid State Age

Laurens Kuiper
CWI, Amsterdam, Netherlands
laurens.kuiper@cwi.nl

Peter Boncz
CWI, Amsterdam, Netherlands
peter.boncz@cwi.nl

Hannes Mühleisen
CWI, Amsterdam, Netherlands
hannes.muehleisen@cwi.nl

***Abstract*—Analytical database systems offer high-performance in-memory aggregation. If there are many unique groups, temporary query intermediates may not fit RAM, requiring the use of external storage. However, switching from an in-memory to an external algorithm can degrade performance sharply.**

We revisit external hash aggregation on modern hardware, aiming instead for robust performance that avoids a "performance cliff" when memory runs out.

To achieve this, we introduce two techniques for handling temporary query intermediates. First, we propose unifying the memory management of temporary and persistent data. Second, we propose using a page layout that can be spilled to disk despite being optimized for main memory performance. These two techniques allow operator implementations to process larger-than-memory query intermediates with only minor modifications.

We integrate these into DuckDB's parallel hash aggregation. Experimental results show that our implementation gracefully degrades performance as query intermediates exceed the available memory limit, while main memory performance is competitive with other analytical database systems.

***Index Terms*—relational databases, database query processing, aggregation**

I. Introduction

Until late in the 20th century, main memory was expensive; therefore, traditional database management systems (DBMS) optimized for disk access, as this was their major bottleneck. "Spillable" data structures like B-trees [1] were used not only to speed up retrieval of persistent data but also inside query operators. As a result, these systems could process workloads that were larger than the small amount of available memory.

Around the 2000s, RAM prices decreased, and database systems optimized for main memory [2], for both persistent data and temporary query intermediates [3]. In these systems, main memory access became the bottleneck, and techniques were devised to make better use of CPU caches [4]. DBMSes have now evolved into large monolithic database servers, often with large amounts of RAM at their disposal.

Pure in-memory systems are not economical, however. Efficient utilization of secondary storage, e.g., by caching, is key to providing good performance at a low cost [5]. In recent years, there has been a renewed interest in buffer management, specifically for solid-state memory, that offers much higher bandwidth and lower latency than magnetic disk [6]–[8]. Data management systems are now reverting to being disk-based without sacrificing in-memory performance [9].

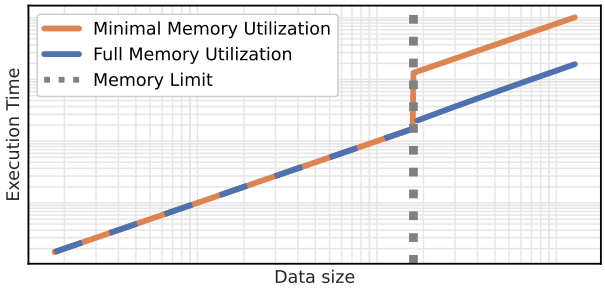

Fig. 1. Conceptual aggregation performance vs data size (log-log scale). When switching from an in-memory strategy to an external strategy that minimizes memory usage, performance degradation is harsh and sudden (a "performance cliff"). A unified strategy for in-memory and external aggregation that utilizes all available memory degrades more gracefully (performance-robust).

This body of research has focused on using storage for persistent data but, for the most part, ignored temporary query intermediates. Analytical (OLAP) systems, which frequently process large volumes of data and often have large query intermediates, became mainstream after DBMSes optimized for main memory. As systems became able to process queries on arbitrary-sized persistent tables, intermediate results can - depending on the query - also grow to arbitrary sizes. In these cases, many modern OLAP systems either abort queries or switch to a traditional disk-based algorithm that is orders of magnitude slower, introducing a "performance cliff", as illustrated in Figure 1.

Given the advancements of OLAP systems in the past two decades [10]–[12], and the research into buffer management on modern hardware [6], [8], OLAP systems should be able to perform more robustly when intermediates exceed main memory. However, traditional buffer managers have fixed-size pages and a statically allocated pool. This inflexibility makes them undesirable for intermediates. Therefore, temporary data is allocated differently. Managing the entire memory pool, i.e., persistent and temporary data, in a cooperative manner may help systems better utilize available memory [13].

In this paper, we go beyond Cooperative Memory Management and take a *unified* approach to memory management for persistent and temporary data. We have developed a specialized page layout specifically for temporary data to accommodate this. We have integrated this into the hash aggregation operator of DuckDB, our in-process analytical

database system. This allows us to efficiently utilize secondary storage, enabling larger-than-memory aggregation for inputs with many unique groups.

Our main contributions are the following:

1) A **unified** approach to memory management that permits variable-size pages and stores pages for temporary query intermediates in the same pool as persistent data, allowing the buffer manager to evict both to storage as needed.
2) A novel **page layout** for temporary query intermediates on buffer pages, optimized for in-memory performance, and spillable to storage without serialization overhead.
3) An integration of the above two techniques into a parallel hash aggregation algorithm that can **gracefully degrade performance** as the memory limit is exceeded.

The rest of the paper is organized as follows. Section II introduces the problem of temporary query intermediates and discusses related work. After presenting Unified Memory Management in Section III, we present our buffer page layout in Section IV. DuckDB's hash aggregation integrates both of these and is presented in Section V. We describe our experimental setup in Section VI, and take a closer look at how Unified Memory Management operates in Section VII. We experimentally evaluate our aggregate implementation and compare it with other implementations in Section VIII. Finally, we summarize and conclude the paper and discuss future research in Section IX.

II. Temporary Query Intermediates

In this section, we discuss the problem of managing temporary query intermediates, as well as discussing related work.

The Memory Hierarchy. Computer storage is organized hierarchically, with smaller capacities and faster access times at the top (CPU registers and Cache Memory) and larger capacities and slower access times at the bottom (Main Memory and Storage). Data access patterns that tend to access data at the top of the hierarchy more than the bottom are favorable, as accessing the higher tiers has a lower latency.

Until late in the 20^{th} century, main memory was expensive, and database systems operated with a very low amount of memory by today's standards. Disk speeds were much lower than today, and storage needed to be accessed frequently due to the low amount of available memory. Disk access was the bottleneck; therefore, database systems were optimized for accessing storage. Around the 2000s, RAM prices decreased, and database systems began optimizing for processing data in main memory [2]. In these systems, memory access has become the bottleneck [4].

Streaming query execution. The common "Volcano" query execution paradigm [14] has a favorable memory access pattern: data is streamed tuple-at-a-time rather than processing query plans operator-at-a-time, allowing tuples to stay at the top of the hierarchy longer. Most modern OLAP systems implement streaming query execution. Their execution engines are usually based on either vectorization, pioneered by VectorWise [10], or data-centric code-generation, pioneered by HyPer [11]. Vectorization processes small vertical chunks of cache-resident vectors at a time. Data-centric code generation processes data such that a tuple is kept in CPU registers as long as possible. Today's systems have greatly improved query processing speeds compared to traditional disk-based systems by keeping data at the top of the memory hierarchy longer, among other reasons.

Blocking Operators. Some relational operators, however, such as aggregation, are *blocking* and cannot output data until all input data has been read. Data cannot be streamed through; Therefore, query intermediates have to be *materialized*, i.e., temporarily kept within the operator. As intermediates grow in size, they move down the memory hierarchy. If intermediates grow such that they no longer fit in memory, they are "spilled", i.e., written to storage to complete the query.

Traditional disk-based database systems implemented relational operators for the low-memory environments that were available at the time. Their operators needed to be prepared to spill; therefore, they often used spillable data structures such as B-trees [1]. Storage access degrades query performance as the cost is significantly higher than main memory access. As the available memory grew, operator design was revisited. Hash-based algorithms that operate fully in main memory perform better than disk-based algorithms [3]. Hash tables, unlike B-trees, are not trivially spillable data structures.

Robustness. Systems that implement hash-based operators may fall back to a disk-based algorithm to complete queries with larger-than-memory intermediates. This choice can be made during query optimization, e.g., using cardinality estimates, but these are often inaccurate [15]. Alternatively, systems can restart a query when intermediates exceed the memory limit at runtime. Either alternative creates a scenario where adding a *single* row to a table could potentially cause the disk-based algorithm to be chosen. Switching algorithms causes a sudden drop in performance, as the hash-based algorithm operates in memory and has $O(1)$ lookup complexity, while the disk-based algorithm accesses disk and has $O(\log n)$ lookup complexity if a B-tree is used.

Operator implementations that adapt to larger-than-memory intermediates at runtime provide more robust query runtimes [16]. Hybrid algorithms such as hybrid hash join [17] realize this with a single, efficient algorithm that works regardless of whether intermediates fit in memory. Hybrid algorithms are limited to an input size less than "the square of memory" [17], which is enough for many use cases.

Memory Management. Traditionally, database systems implement a buffer manager to move paged data between main memory and storage. Operators only specify when and for how long pages are needed, and the buffer manager will fetch them from memory or disk. Common wisdom is to use a buffer manager with fixed-size pages and a fixed-size pool, the size of which is user-configured. The fixed size makes them unattractive for temporary query intermediates because large objects such as hash tables cannot be stored on a single page. Spreading large objects over multiple pages makes using them less efficient and more complex [9]. Therefore, temporary query intermediates are usually allocated differently [13].

This essentially creates two memory pools, one for persistent data on fixed-size pages and one for temporary data that allows variable-size allocations. The fixed-size buffer pool cannot shrink if more memory is needed for intermediates; therefore, the size must be tuned to the workload. Cooperative memory management [13] challenges this, but to the best of our knowledge, this has yet to be implemented in any system. Allocating from a different pool also means that the buffer manager is not used to offload intermediates, but operators must explicitly read from and write to a temporary file.

When query intermediates are spilled from memory to storage, random access time is greatly increased. In storage, data is no longer byte-addressable but block-addressable. New technologies such as byte-addressable storage or persistent memory may change this in the future, but these are not generally available. Going from byte-addressable to block-addressable requires changes to operator algorithm design: blocks must be loaded into memory before the data can be randomly accessed, e.g., by a hash table.

MMAP. One option to circumvent this problem would be to use memory-mapped (MMAP) files, an operating system (OS) feature that maps the contents of a file on secondary storage into a program's address space, making data that resides in storage byte-addressable. MMAP essentially allows the database system to keep using the same algorithm that assumes everything fits in memory by offloading the responsibility of loading and evicting pages to the OS. With this approach, buffers can exceed the available memory limit, which allows, e.g., hash table probes to randomly access data in storage.

MMAP requires the OS, which has no knowledge of the workload, to evict and load pages. Randomly accessing storage is orders of magnitude slower than randomly accessing data in memory and is unlikely to provide robust query performance. Furthermore, the OS's page eviction mechanisms were found not to scale beyond a few threads for larger-than-memory DBMS workloads on high-bandwidth secondary storage devices [7]. This also effectively rules out the use of "memory rewiring" [18] for larger-than-memory processing on many-core architectures because it uses MMAP.

SSDs. Not long after database systems optimized for main memory, the use of Solid-State Disks (SSDs) became mainstream because of their lower latency and higher throughput. The potential performance benefit of SSDs was quickly recognized in the database literature [19]. Advancements in hardware, as well as advancements in optimizing for SSDs, have ultimately led to modern SSD-optimized storage managers such as LeanStore [6]. However, these advancements have focused on using SSDs mainly for persistent storage, not temporary query intermediates. This is partly due to many data management systems having evolved into large monolithic servers running in high-memory environments.

External Processing for OLAP. There is a clear need for analytical data management in more economical environments [20]. Many of the developments discussed in this section took place before OLAP became popular, which was in the late 1990s and early 2000s, after main memory database systems had already become mainstream. As a result, OLAP systems do not have a long history of processing larger-than-memory intermediates like transactional (OLTP) systems do. Most workloads fit in main memory, but the user experience can often be frustrating if they do not. When the OLAP system runs out of memory to complete a query, it either aborts or switches to a much slower traditional disk-based algorithm. The former leaves the user unable to complete the query, and the latter results in unpredictable query runtimes.

Rather than an all-or-nothing approach to memory usage, we argue that analytical database systems should be able to efficiently use *all* available resources to complete a query to provide robust query runtimes and graceful performance degradation if the memory limit is exceeded. We should prioritize the top of the memory hierarchy: use memory when possible and only use storage if necessary. To achieve this, we propose using Unified Memory Management.

III. Unified Memory Management

In this section, we present Unified Memory Management, a buffer manager that unifies memory management for persistent and temporary data and permits variable-size allocations.

Persistent Data. DuckDB does not allocate a fixed-size buffer pool for persistent data for two reasons:

1) As identified in the previous section, a fixed-size pool occupies memory that could potentially be used for temporary query intermediates.
2) DuckDB is not a server but an in-process DBMS. Its allocations live within the same address space as the host process. To avoid interfering with the host process, it is important that DuckDB's resource consumption is low when idle; therefore, memory must be deallocated.

For the reasons mentioned in the previous section and portability reasons, DuckDB also does not use MMAP. Each buffer is allocated individually.

To avoid fragmentation, DuckDB uses a fixed page size of $2^{18} = 262{,}144$ bytes (256 KiB) in storage; therefore, all pages for persistent data are of this size. This page size is chosen for OLAP workloads, which is DuckDB's main use case. 256 KiB is 64 times larger than the default page size of 4 KiB used in Cooperative Memory Management and most OLTP systems. If a page is no longer needed, it is added to the eviction queue, which is a lock-free concurrent priority queue with a least-recently-used (LRU) policy. Buffers in the eviction queue are evicted when newly requested memory would cause the memory limit to be exceeded. If enough memory is available, persistent data stays cached in memory. If the newly requested allocation has the same size as a page in the queue that can be evicted, the buffer is reused.

Temporary Data. Allocations for temporary data are more flexible. DuckDB's buffer manager distinguishes three types of temporary allocations: 1) Non-paged allocations, 2) Paged fixed-size allocations, and 3) Paged variable-size allocations. *Non-paged allocations* are non-spillable allocations of any size. Despite being non-paged, the allocation goes through the buffer manager so that it may decide to evict other pages if the

allocation would cause the memory limit to be exceeded, as is the case for Cooperative Memory Management. If the allocation is no longer needed, it is deallocated. *Paged fixed-size allocations* have the same size as persistent pages (256 KiB) and can be efficiently swapped in and out of a temporary file in storage if needed. Note that the temporary file is completely separate from the database file that stores persistent data and could potentially reside on a different storage device if desired. Having the same size for temporary and persistent pages allows buffers to be reused. *Paged variable-size allocations* can also be written to storage, but because it is of variable size, each is written to a separate temporary file.

Non-paged allocations and paged variable-size allocations are used sparingly only if efficient query processing requires it, e.g., for the buckets of a hash table or strings larger than 256 KiB. Paged fixed-size allocations are the most common and are used to store almost all temporary query intermediates, even if ample memory is available. In our operator implementations, we try to eagerly destroy temporary pages as soon as they are no longer needed: this deallocates the memory if the page was loaded or frees up disk space if the page was spilled. This prevents the total space used for intermediates (memory + disk) from exceeding the necessarily required space.

Because allocation performance can affect query performance [21], we are cautious about how the design of the buffer manager affects allocation performance. A micro-benchmark is performed in Section VII.

Cooperative Memory Management. The proposed buffer manager allows all available memory to be used for persistent and temporary data, rather than reserving and managing these separately, like Cooperative Memory Management [13]. The key difference, however, is that, unlike Unified Memory Management, Cooperative Memory Management does not have *paged temporary memory*. Because Unified memory management has paged allocations for temporary data, loading persistent data, as well as more allocations for temporary data, can cause not just persistent data but also temporary data to be evicted. Note that eviction only occurs when memory is full. Another important difference is that allocated pages for persistent data are reused for temporary data and vice-versa.

Compatibility. The proposed buffer manager has a familiar API with methods for (un-)pinning pages, similar to those in other systems. However, it does not support the notion of *dirty* pages because DuckDB uses lightweight compression [22] to compress its columnar storage. Hence, it is not generally possible to perform in-place updates, as pages are always fully rewritten. The proposed buffer manager is, therefore, compatible with other systems with minor modifications.

IV. Page Layout

In this section, we first discuss the considerations that went into designing DuckDB's page layout for temporary data before giving an overview of the implementation.

DSM vs. NSM. There are many ways to lay out relational data on fixed-size pages. Traditional database systems such as PostgreSQL store data on these pages in row-major format also called the N-ary Storage Model (NSM), which has the advantage of colocating *tuples* in memory, improving the locality of accessing tuples. Page layouts like PAX [23] store data in column-major format, also called the decomposition storage model (DSM), which has the advantage of colocating *attributes* in memory, improving the locality of accessing columns. Both layouts are used for persistent data but could, in theory, also be used for temporary data.

For intermediates, a row-major layout was shown to be optimal, even in column-major systems, for join and aggregate hash tables [24], as well as for sorting [25]. Co-locating a tuple's attributes in memory improves the locality of accessing subsequent attributes of the same row for comparisons, reducing cache misses. Using fixed-sized rows over variable-sized rows allows efficient access into these attributes using offsets, further improving comparison performance [24]. Our goal is to achieve good in-memory performance, as most workloads fit in main memory; therefore, we choose to use fixed-size rows in our page layout.

Variable-Size Data. Using fixed-size rows, however, means that variable-sized data types such as strings, which are omnipresent in real-world data sets [26], and arrays cannot be placed within the rows. The variable-sized types are stored elsewhere and can be addressed implicitly, e.g., with an offset, or explicitly, with a pointer stored within the fixed-size row. Implicit addressing requires additional information, e.g., the ID of the page where the variable-sized data is stored, to locate the relevant data. This causes more indirection and is less efficient than explicit addressing. Furthermore, implicit addressing complicates arbitrarily jumping between rows stored on different pages, e.g., for a bucket-chained hash table, as the page ID of the variable-sized data may differ between rows. Therefore, we will only consider explicit addressing, as this does not compromise in-memory performance.

The actual variable-sized data is stored outside of the row. If it is stored in a global pool that resides in memory, the pool itself could exceed the available memory limit, causing the query to be aborted. MMAP would allow such a global pool to be spilled to storage, but we do not use it for the reasons mentioned earlier. Variable-sized data should, therefore, also be stored on pages to enable spilling. It can be stored either on the same or a different page. If stored on the same page, the fixed-size rows could grow from the top, while the variable-sized data grows from the bottom. This can lead to inefficient use of pages, however: If a page has room for more fixed-size rows, but these rows have long strings that do not fit, we move on to the next page, leaving the space on the previous page unoccupied. Therefore, we store fixed-size rows and variable-size data on different pages.

(De-)Serialization. Explicit addressing for variable-sized data in combination with storing variable-sized data on buffer pool pages creates a problem of invalid references: If pages storing variable-size data are evicted and loaded back into memory, their addresses may change, invalidating the explicit addresses in the fixed-size rows pointing to this data. A common way to address such issues is to serialize the data when it

is written to storage and deserialize when it is read back into RAM. However, (de-)serialization can easily dominate query execution time if not implemented efficiently [27]. Given the current SSD speeds, any (de-)serialization method will cause spilling to become CPU-bound rather than I/O-bound.

Arrow Flight [28], an efficient data (de-)serialization format, addresses this problem by only requiring serialization on write and little to no deserialization on read. However, Arrow Flight stores data in column-major format; therefore, using it as our layout would compromise the in-memory performance of our blocking operators [24]. Arrow flight also has a header for each batch, which we do not need as the columns and types of query intermediates are already known. A format like Arrow Flight does not yet exist for row-major data.

Serializing before writing, besides being potentially costly itself, also requires the operator to explicitly serialize pages when they are unpinned, as the buffer manager may evict them at any time. This may result in unnecessary (de-)serialization if the page is not offloaded after all. Alternatively, if the buffer manager is aware of the content that is stored on the pages, it can serialize the page before evicting. This has the benefit of only serializing the data when it absolutely needs to: only when the data is actually written to storage. However, this requires the buffer manager to know about the content of pages, which will complicate the design and implementation.

Requirements. This discussion clearly shows that using current page layouts for query intermediates will compromise in-memory performance. This is undesirable, as most workloads fit in memory. Using a different page layout for pages in memory and pages in storage will lead to ungraceful performance degradation as (de-)serializing data is costly. This establishes the need for a page layout that is *both* efficient in memory and can be spilled to storage. We have determined the following requirements. The page layout must:

1) Use a row-major data representation with fixed-size rows;
2) Store variable-size data on separate pages;
3) Use explicit addressing for variable-size data;
4) Be spillable to storage without additional serialization.

In the remainder of this section, we present the design of DuckDB's page layout for temporary data. Each tuple is represented by a fixed-size row of which the types, and, therefore, the widths and offsets of each attribute, are known when the query plan is generated. This information is stored once, globally, rather than once per page. Fixed- and variable-size data are stored on separate pages. This fulfills requirements 1 and 2. In the remainder of this section, we explain how requirements 3 and 4 are fulfilled.

Variable-Size Row. DuckDB uses the string type proposed by Umbra [9], which is 16 bytes. The first 4 bytes store the length of the string. Small strings of 12 characters or less are *inlined* in the remaining bytes. For *non-inlined* strings longer than 12 characters, a 4-byte prefix and a pointer is stored. As discussed, a row-major format colocates a tuple's attributes in memory, improving cache efficiency when comparing them subsequently. However, for non-inlined strings, we have to follow the pointer, which may cause a cache miss. To improve locality, we store subsequent non-inlined strings in a row sequentially as well. In other words, each fixed-size row has a corresponding variable-size row.

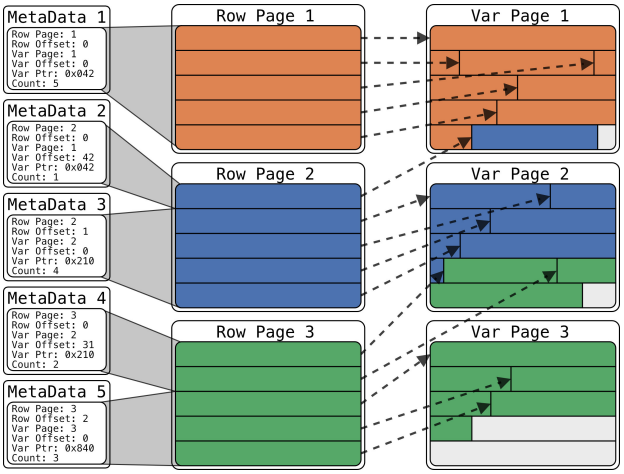

Fig. 2. DuckDB's page layout for fixed-size rows and corresponding variable-size rows. By using a fixed page size, the row and var pages will not have a one-to-one relation with each other, and a small amount of metadata is needed to describe how they line up. The metadata always describes tuples that have their data stored on the same row and var pages. If a row or var page is full, another metadata segment is created. This ensures that all variable-size data within a metadata segment have the same base pointer, allowing pointer recomputation to be done per segment.

Pointer Recomputation. As discussed, if a page storing variable-size data is spilled to disk, it will be loaded into a different location in memory. The pointers in the fixed-size rows, however, still point to the previous location. These can be *recomputed* if the previous base pointer of the page is stored: For example, consider a string stored at address `0x48` in a page with base pointer `0x42`, i.e., the string is stored at offset `6` in the page. The page that stores the string is then spilled and loaded back into memory, and now the page has a base pointer of `0x500`. We can recompute the new pointer by subtracting the previous base pointer from the stored pointer and adding the new base pointer: `0x48 - 0x42 + 0x500 = 0x506`. This can be done in place and *lazily*, i.e., only after we have detected that the variable-size data page has actually gone to disk instead of pre-emptively.

Recomputing the pointers within a fixed-size row requires knowing the previous base pointer of the page that stores its corresponding variable-size row. However, storing that pointer within each row introduces an 8-byte overhead. Instead, we can exploit the fact that many subsequent fixed-size rows will have their corresponding variable-size row stored on the same variable-size data page. We could even go as far as allocating one variable-size data page per fixed-size data page, but this requires allocating a non-standard page size to store the variable-size data, which we would like to prevent. If possible, we prefer to allocate the standard page size.

Using the standard page size means that the fixed- and variable-size row pages will not line up. This is illustrated on the right-hand side of Figure 2. Here, many subsequent fixed-

size rows have their corresponding variable-size row stored on the same variable-size data page; therefore, storing a pointer in each row would be wasteful. Instead, we store a small amount of in-memory metadata, as shown on the left-hand side of Figure 2, that describes how the fixed- and variable-size rows on the different pages line up. This allows us to recompute the pointers without an overhead of 8 bytes per row. The explicit addresses are recomputed only if the pages have been spilled to disk, which we can detect by comparing the stored pointer with the current page pointer; therefore, the performance in RAM is unaffected.

The proposed page layout can be implemented in any system that uses explicit paged I/O, as it is only a layer of metadata on top of pages.

V. Robust External Hash Aggregation

In this section, we first discuss the considerations that went into designing DuckDB's hash aggregation before presenting the implementation.

Aggregation is a key operator for OLAP workloads; therefore, its performance is stressed in analytical benchmarks such as TPC-H [29]. If the input data is pre-sorted on the group keys, aggregation can be performed in a streaming fashion [30], but otherwise, aggregation is a blocking operator. Grouped aggregation can be resolved by sorting the input data, but if the data fits in memory, hash-based algorithms give better performance [2]. Hashing also has the theoretical time complexity advantage of $O(n)$ rather than $O(n \log n)$. For these reasons, most OLAP systems implement hash aggregation. Therefore, we also implement hash aggregation, as we aim not to impair main memory performance.

Parallelism. Modern hardware has many available CPU cores. Utilizing them well is essential for aggregation performance. *Plan-driven* parallelism [31] decides a static thread count when planning a query, then lets each thread execute fragments of the query plan. Fragments are connected with a parallelism-aware *exchange* operator that re-routes tuples to different plan fragments. Other operators are kept largely unaware of parallelism; therefore, this approach simplifies operator design. However, re-routing tuples causes overhead, and workloads cannot be balanced if data distributions are skewed [12]. *Morsel-driven* parallelism [12] addresses this with a framework for parallelism that schedules fine-grained tasks on small fragments of input data. Tuples need not be re-routed, and workloads are more evenly balanced across threads. However, operators are required to be parallelism-aware, complicating their design. Due to its large performance benefits, DuckDB implements morsel-driven parallelism.

In morsel-driven parallelism, data is processed in pipelines. Within a pipeline, data moves from an input, such as a table scan, through streaming operators, e.g., projections, to a "pipeline breaker", i.e., a blocking operator, such as hash aggregation. Pipelines themselves are parallel: threads work concurrently on the same pipeline. Operators may have a local state per thread and one state shared across all threads. Cross-thread communication (or even cross-socket communication in the case of NUMA) causes overhead; therefore, accessing the shared state should be done sparingly.

Low Cardinality Aggregation. Many aggregations reduce the size of the input to just a few rows. A typical example is TPC-H query 1, which reduces the input to just four rows, regardless of the scale factor. Low cardinality aggregations are trivial to perform in parallel in morsel-driven parallelism, with ungrouped aggregation being an extreme example. Thread-local pre-aggregation is performed in parallel, reducing the input to a few rows per thread or a single row if the aggregation is ungrouped. After all input data has been read, the data from each thread is combined. Even if there are many threads, combining, e.g., four rows from each thread, has a negligible cost compared to the thread-local pre-aggregation, which could have aggregated millions of rows. Therefore, combining can be done by a single thread without impairing performance.

High Cardinality Aggregation. Aggregation is not limited to low cardinality outputs. Large data volumes can have a large number of unique groups in the output. This leads to high cardinality aggregations: checking whether a column is a primary key, if this is not enforced by the data format, eliminating duplicate rows in machine learning data sets, queries with `DISTINCT`, or grouping by unique customer in a large customer base, to name a few.

When the output has many rows, performing parallel aggregation is nontrivial. After thread-local pre-aggregation, threads have collected large amounts of data. In the extreme case where each group in the input occurs exactly once, the data has not been reduced at all, and potentially millions of rows remain. Compared to low cardinality aggregation, combining all thread-local data now has a significant cost, and performing it with a single thread impairs performance.

The authors of morsel-driven parallelism take an approach similar to IBM's DB2 BLU [32] and perform thread-local pre-aggregation in a small fixed-size hash table. When this hash table is full, the data is partitioned. After all data has been pre-aggregated and partitioned, the second phase begins. Partitions are exchanged between threads and aggregated independently in parallel. After a thread finishes aggregating a partition, its tuples are immediately pushed into the next pipeline before processing any other partitions. By keeping the thread-local hash table small and fixed-size, fewer cache misses are incurred, and costly hash table resizes are prevented.

Data Distributions. This approach to aggregation efficiently reduces heavy hitters in skewed data distributions and exploits *interesting orderings* found in real-world data [33], such as many of the same group keys appearing in succession. Because the data is partitioned *after* reducing, this approach to parallel aggregation is more robust to skewed distributions than exchange-based parallelism, which partitions data across threads *before* reducing, creating imbalanced workloads.

A downside, however, is that groups can be added to thread-local hash tables multiple times if they appear at intervals that are larger than the fixed-size hash table, e.g., in random uniform distributions with many unique groups. This causes memory consumption to grow linearly with the input size

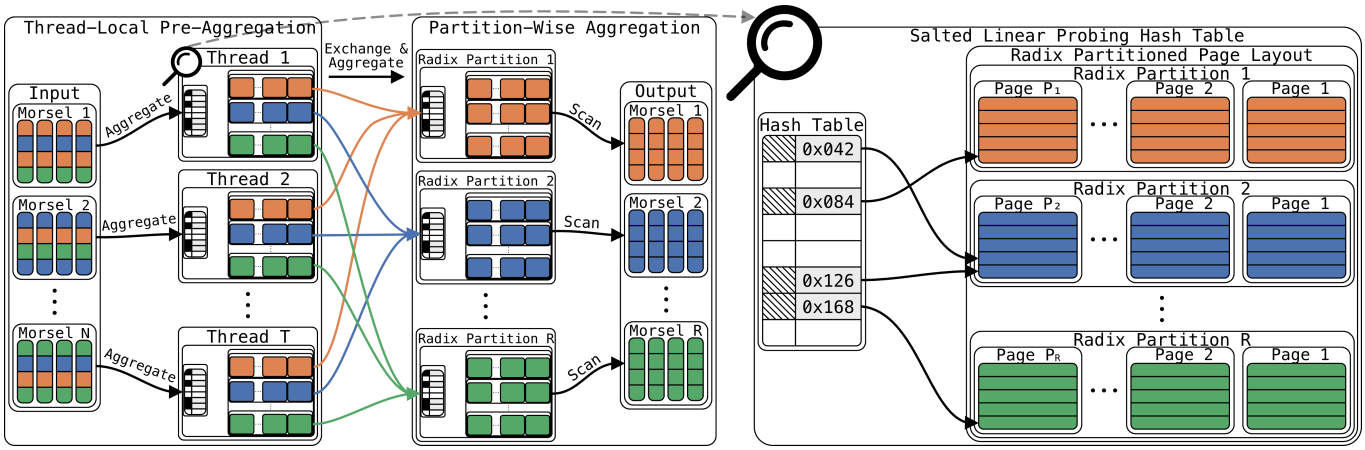

Fig. 3. DuckDB's hash aggregation. Morsels are assigned to threads until all input data has been read. During phase one, each thread pre-aggregates data in a small fixed-size linear probing hash table with one level of indirection, i.e., offsets obtained from hashes access an array of pointers pointing to tuples. Tuples are radix partitioned and stored using DuckDB's spillable page layout, enabling larger-than-memory aggregation. After pre-aggregation, partitions are exchanged and aggregated partition-wise in parallel. Fully aggregated partitions are immediately scanned, effectively becoming morsels in the next pipeline.

rather than with the output size. Resizing the hash table could prevent this, but this would impair the performance of the more common case of non-uniform real-world data, as it results in more cache misses. Plan-driven parallelism does not suffer from this problem, as the exchange operator always routes the same groups to the same threads, but, as explained, plan-driven parallelism does not perform well in case of skew, which is common in real-world data sets [26].

External Aggregation. High cardinality aggregations may not fit in RAM on limited hardware or with huge datasets; therefore, intermediates must be spilled to storage to complete the aggregation. As mentioned in the previous section, this can lead to a frustrating user experience when using OLAP systems, as these often only perform well if intermediates fit into memory. If intermediates do not fit, query performance is not robust, and some queries may not finish at all, as we will see in Section VIII.

We now present DuckDB's *embarrassingly external* hash aggregation, which integrates the proposed page layout. This aggregation method is inspired by HyPer's [12] and IBM DB2 BLU's aggregation [32], with key performance optimizations. We describe a partitioned aggregation approach that, as we will see, can trivially process larger-than-memory intermediates without explicitly writing data to storage, leaving the responsibility of spilling to the buffer manager, thereby simplifying operator design. Our design is illustrated in Figure 3.

In the first phase, *Thread-Local Pre-Aggregation*, we assign morsels to threads until all input data has been read. There are usually many more morsels than threads. Data is scanned from morsels in batches of up to 2,048 tuples. Threads pre-aggregate in a small, fixed-size ($2^{17} = 131{,}072$) hash table. The hash table itself consists of two parts: 1) An array of 64-bit entries, which contains a pointer in the lower 48 bits, pointing to the corresponding entry, and 2) temporary pages for hash table entries consisting of groups and aggregates as well as the corresponding pages for variable-length values.

Salt. Pointers have a width of 64 bits on 64-bit CPU architectures, but only the lower 48 bits are used, as this allows for up to ≈ 281 TB of address space. We use the remaining 16 bits of the pointer to store the upper 16 bits of the hash of the corresponding tuple, which we call *salt*, shown in the *Hash Table* of the *Salted Linear Probing Hash Table* in the right-hand side of Figure 3. Indirect access to the hash table, i.e., storing entries that point to tuples rather than storing tuples directly, keeps the area that is randomly accessed small and allows for specific optimizations, which will be explained in this section, that would not be possible otherwise.

Collision Resolution. As usual, the offset into the array of 64-bit entries is obtained from the lower bits of the hash. Collisions are resolved using linear probing. Before following the pointer to the tuples and comparing group keys, we first compare the salt. Only if the salt is equal do we follow the pointer to compare group keys. For uniform hash functions, the salt effectively reduces the chance of having to compare group keys of tuples that do not match by a factor of $2^{16} = 65{,}536$. This is most effective with linear probing. In a bucket-chained hash table, the salt requires either a bitwise OR, which would flip more bits and, therefore, increase the chance of false positives, or a 16-bit bloom filter that uses just 4 bits of the hash. Usually, performance degrades as the hash table fills up due to increased collisions, causing more group key comparisons and more random access. This optimization allows almost all collisions to be resolved much more efficiently.

Partitioning. Rather than partitioning tuples when the hash table is reset like IBM DB2 BLU [32] and HyPer [12], our implementation directly materializes tuples into partitions that use the proposed page layout. By materializing tuples directly into partitions, we avoid copying tuples more than once. Note that the partitioned data is in row-major representation, while the incoming data is in column-major representation: the conversion of column-major to row-major takes place simultaneously while partitioning the data. This optimization

is possible because the hash table indirectly accesses tuples as shown in the *Radix Partitioned Page Layout* of the *Salted Linear Probing Hash Table* in the right-hand side of Figure 3.

The partitions are determined by radix, i.e., a few of the middle bits of the hash that were not yet used for the salt (upper bits) or for the offset into the hash table (lower bits). It is important that any of the used bits do not overlap, as this would lead to more collisions and/or reduced effectiveness of the salt. The number of bits used to partition depends on the number of active threads and whether the size of the intermediates is larger than memory, as will be explained later. Because the data is partitioned *after* pre-aggregation rather than before, partitions are of roughly equal size.

RAM-Oblivious. We reset the hash table once it is two-thirds full. This threshold was experimentally determined. Only the array of 64-bit entries is reset while the tuples stay in place; therefore, resetting is an inexpensive operation. The pages that store these tuples can now be unpinned, as the tuples are no longer active in the hash table. With this approach, partitions are never explicitly written to storage by the aggregation operator, which implementations like IBM DB2 BLU's aggregation [32] would do in low-memory situations. Instead, the buffer manager writes individual pages, rather than entire partitions, to storage when needed. We analyze the spilling behavior of our implementation in-depth in Section VII.

This simplifies operator design, as the aggregation operator is largely unaware of which pages are in memory and which are on disk. Similar to how cache-oblivious algorithms [34] are oblivious to the size of the CPU caches, our algorithm is virtually *RAM-oblivious* during the *Thread-Local Pre-Aggregation* phase: the algorithm's behavior does not depend on the memory limit. The only requirement is that the small fixed-size hash table fits in memory.

During the second phase, *Partition-Wise Aggregation*, the memory limit matters, as the question becomes whether one fully aggregated partition per thread fits in memory. This can be achieved by over-partitioning, i.e., creating many more partitions than threads. This keeps the memory pressure low during the second phase as well. When a partition is fully aggregated, its results are immediately pushed to the next operator in the pipeline, freeing up the used pages.

The design of DuckDB's hash aggregation operator should degrade performance gracefully as the size of the temporary query intermediates exceeds the available memory limit, and only the pages that do not fit are spilled to storage. Our aggregate implementation has been part of every DuckDB release since version 0.9.0 and is widely used[1].

VI. Experimental Setup

In this section, we describe our experimental setup. All of our experiments are run on AWS EC2 using Ubuntu 22.

Hardware. We choose the `c6id.4xlarge` instance because its specifications are similar to affordable hardware such as a laptop. This instance has an Intel Xeon 8375C CPU with 8 cores (16 threads), 32 GB of DDR4 RAM, and 1 TB of NVMe storage. We set the tenancy of the instance to `dedicated` so that the entire node is reserved, but we do not use the rest of the node's capacity. We also use the available Instance Storage, which is physically attached to the host, unlike the usual Elastic Block Storage on EC2. This setup eliminates the *noisy neighbor* problem that cloud environments may have; therefore, our results are more consistent.

Data. We create a grouping benchmark using the data generator from TPC-H, which is incorporated in DuckDB's `tpch` extension. We generate the `lineitem` table at scale factors 1, 2, 4, 8, 16, 32, 64, and 128. At these scale factors, the row count ranges from 6,001,204 to 768,046,921, and the size of the generated CSV ranges from 0.72 GB to 96.72 GB.

Query. Although aggregation can easily dominate the runtime of a query, *isolating* the performance of any relational operator in a benchmark is difficult because we can often only reliably observe end-to-end query runtime. This is the case even if systems have a query profiler because the execution of multiple operators is interleaved in streaming query engines. Measuring end-to-end query runtime introduces unwanted and unrelated overheads, e.g., small overheads such as parsing and optimizing, scanning base tables, and transferring the result set through a client protocol. The latter is especially costly for large result sets, e.g., high cardinality aggregations, and can easily dominate query execution time [27].

Instead of transferring the result set through a client protocol, we could write the result to a table instead, i.e., `CREATE TABLE ... AS`. This is slightly better but also introduces a large amount of overhead in the form of bulk insertion, which may have a large performance difference across different systems. Adding a cheap, ungrouped aggregate on top of the expensive aggregate that we want to measure would be even better, but in some cases, this allows query optimizers to remove unused columns, potentially leading to unequal plans across different systems. Therefore, we have devised the following benchmark query:

```
SELECT group_key1, group_key2,
       ..., group_keyG,
       ANY_VALUE(col1), ANY_VALUE(col2),
       ..., ANY_VALUE(colC)
FROM lineitem
GROUP BY group_key1, group_key2,
         ..., group_keyG
OFFSET N - 1;
```

Here, `N` is the number of unique groups in the query, which has been precomputed for each grouping. This yields a result set of exactly one row. Technically, this query is underspecified, so any single row of the input satisfies the query. However, the number of unique groups is not known beforehand; therefore, systems are forced to fully process and discard the first `N - 1` groups before emitting a single row. Additional columns other than group keys are selected using the `ANY_VALUE` aggregate function to increase the memory pressure without changing the number of unique groups.

[1]DuckDB source code can be found at https://github.com/duckdb/duckdb

TABLE I
OUTPUT SIZE OF GROUPING LINEITEM BY DIFFERENT COMBINATIONS OF COLUMNS. AN OUTPUT SIZE OF 25.00% MEANS THAT THE NUMBER OF ROWS IN THE OUTPUT IS EQUAL TO 0.25 TIMES THE INPUT SIZE.

Number	Grouping	Size
1	l_returnflag, l_linestatus	4 rows
2	l_partkey	3.33%
3	l_partkey, l_returnflag, l_linestatus	10.58%
4	l_suppkey, l_partkey	13.33%
5	l_orderkey	25.00%
6	l_orderkey, l_returnflag, l_linestatus	34.87%
7	l_suppkey, l_partkey, l_returnflag, l_linestatus	36.17%
8	l_suppkey, l_partkey, l_shipinstruct	45.34%
9	l_suppkey, l_partkey, l_shipmode	61.83%
10	l_suppkey, l_partkey, l_shipinstruct, l_shipmode	88.56%
11	l_orderkey, l_partkey	99.99%
12	l_orderkey, l_suppkey	99.99%
13	l_suppkey, l_partkey, l_orderkey	100.00%

Groupings. We group the `lineitem` table by different combinations of columns, shown in Table I. We have two variants of each grouping. The *thin* variant selects only the group columns. The *wide* variant selects all other columns using the `ANY_VALUE` aggregate function.

With these groupings, the thin and wide variants, and different scale factors, this benchmark represents a wide variety of aggregations, ranging from low to high memory pressure. For example, the lowest memory pressure is the thin variant of grouping by `l_returnflag`, `l_linestatus`, which only materializes four rows of two columns. The highest memory pressure is the wide variant of grouping by `l_suppkey`, `l_partkey`, `l_orderkey`, which essentially materializes the entirety of the `lineitem` table. All of our experiments are publicly available on GitHub[2].

VII. LOADING, SPILLING & ALLOCATING

In this section, we take a closer look at how the proposed unified memory management operates.

Loading & Spilling. We first examine how DuckDB's buffer manager decides to load and spill pages when running an aggregation workload. The workload we have selected for this experiment is thin grouping 4 at scale factor 128 because it requires materializing a fair amount of intermediate data but not so much that the entire aggregation becomes bottlenecked by I/O. This grouping only selects `l_orderkey`, which is just over 1.5 GB in DuckDB's file format, which uses lightweight compression techniques [22]. We consider two scenarios: In the first scenario, a single connection runs this grouping 10 times in a row, with DuckDB configured to have 4 threads and a memory limit of 3.5 GB. DuckDB is rarely used as a server: a single connection is a common use case for local data set analysis. We have chosen a memory limit of 3.5 GB, as this is approximately the total size of the intermediates needed for this grouping and, therefore, necessitates evicting pages. This memory limit yields more interesting results than a high or low memory limit, as, in theory, not much I/O is needed to complete the query, but wrong decisions by the buffer manager may cause unnecessary I/O. In the second scenario, four connections simultaneously run this grouping 10 times in a row, with DuckDB configured to have $4 \times 4 = 16$ threads and a memory limit of $4 \times 3.5 = 14$ GB.

As mentioned in Section III, DuckDB's *eviction policy* is to evict pages using a concurrent LRU queue. Pages of all types are added to the same queue: persistent, temporary, fixed- and variable-size; no distinction is made between the types of pages. Eviction policies have been researched in-depth in the context of traditional buffer pools, which only store persistent data but not in the context of temporary data. These policies try to keep often-used persistent data in memory to speed up access. Temporary data is short-lived; therefore, having a different policy for these pages may be beneficial. We will refer to DuckDB's default eviction policy as the *Mixed* policy. For this experiment, we implement two more eviction policies, which store persistent and temporary pages in two separate LRU queues. *TemporaryFirst* evicts temporary pages before evicting persistent pages. *PersistentFirst* evicts persistent pages before temporary pages. We show the results of this experiment in Figure 4.

With a single active connection, Mixed, TemporaryFirst, and PersistentFirst have similar respective execution times of 69.8s, 72.1s, and 66.8s. The size of the temporary file was observed to stay under 500 MB for PersistentFirst and under 2 GB for TemporaryFirst and Mixed. Persistent data is read at the start of each query during thread-local pre-aggregation. Only temporary data is needed during partition-wise aggregation once all data is materialized. Evicting persistent data does not require writing to storage because it is already stored in the database file. Evicting temporary data, on the other hand, requires writing it to storage and is, therefore, more costly. As we can see, this results in PersistentFirst being the most efficient strategy when a single connection is active.

With four active connections, Mixed, TemporaryFirst, and PersistentFirst have varied respective execution times of

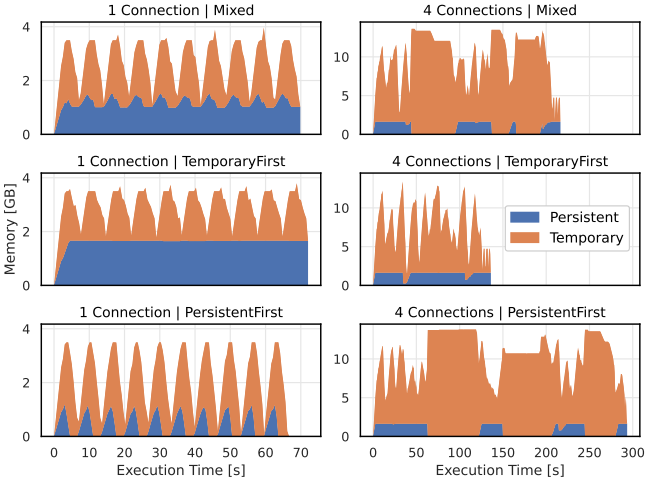

Fig. 4. Visualization of DuckDB's memory when repeatedly performing thin grouping 4 at scale factor 128 in a single-connection and multi-connection scenario with different buffer eviction policies.

[2]Experiments: https://github.com/lnkuiper/experiments/tree/master/oocha

TABLE II

EXECUTION TIME IN SECONDS FOR THE THIN VARIANT OF ALL GROUPINGS AT SCALE FACTORS 2, 8, 32, AND 128. LOWER IS BETTER. THE LOWEST EXECUTION TIMES ARE HIGHLIGHTED IN BOLD. 'A' DENOTES THAT THE QUERY WAS ABORTED. WE SUMMARIZE BY NORMALIZING EXECUTION TIMES TO DUCKDB AND THEN TAKING THE GEOMETRIC MEAN.

SF	2				8				32				128			
System	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um
Grouping 1	**0.01**	0.08	**0.01**	0.02	**0.03**	0.27	0.05	0.04	**0.14**	1.10	**0.14**	0.16	**0.54**	4.06	**0.54**	A
2	0.08	**0.04**	0.13	**0.04**	0.44	**0.16**	0.66	0.19	2.03	0.75	2.86	**0.68**	10.80	**4.04**	12.83	A
3	**0.13**	0.30	0.20	**0.13**	0.58	1.21	1.00	**0.51**	2.78	6.35	4.28	**2.24**	**14.49**	42.47	348.10	A
4	0.12	0.08	0.16	**0.07**	0.58	0.29	0.83	**0.28**	2.86	1.63	3.38	**1.22**	22.06	**9.80**	231.86	A
5	**0.05**	0.08	0.07	**0.05**	0.18	0.29	0.36	**0.16**	0.74	1.57	1.66	**0.54**	**3.17**	9.41	6.86	A
6	**0.08**	0.35	0.21	0.15	**0.30**	1.42	0.81	0.55	**1.27**	7.32	3.27	2.12	**5.80**	48.79	213.41	A
7	**0.17**	0.37	0.28	0.23	**0.72**	1.56	1.36	0.87	**3.42**	8.32	5.61	4.32	**24.62**	51.57	457.52	A
8	0.23	0.36	0.27	**0.20**	0.97	1.51	1.39	**0.78**	5.25	8.11	5.72	**3.44**	65.97	**49.49**	412.86	A
9	**0.16**	0.37	0.30	0.21	**0.74**	1.53	1.58	0.78	3.59	8.01	6.42	**3.47**	**32.77**	46.51	444.50	A
10	**0.25**	0.50	0.41	0.38	**1.10**	2.06	1.96	1.50	**5.78**	11.34	169.63	14.98	89.27	**77.27**	576.68	A
11	**0.13**	**0.13**	0.37	0.18	0.59	**0.51**	1.71	0.65	2.72	**2.59**	6.94	2.86	21.38	**18.43**	413.54	A
12	**0.13**	**0.13**	0.36	0.18	0.59	**0.52**	1.73	0.65	2.65	**2.59**	6.80	2.84	21.89	**18.22**	411.58	A
13	**0.15**	0.17	0.38	0.22	**0.64**	0.70	1.80	0.79	**2.87**	3.64	7.24	3.39	**22.20**	28.31	432.33	A
Geometric Mean Normalized to DuckDB	**1.00**	1.69	1.80	1.13	1.00	1.53	1.98	**0.98**	1.00	1.69	2.17	**0.94**	**1.00**	1.48	8.74	A

216.4s, 135.9s, and 293.5s. This is the opposite order of the previous scenario. The size of the temporary file was observed to stay under 8 GB for all three policies. In the figure, it appears like all persistent data is evicted at some point, but some of it is always loaded into memory, although the amount is so small that it is not visible. When most persistent data is evicted, thread-local pre-aggregation slows down because every read requires accessing storage. This behavior causes memory to clog up with temporary data, and overall query throughput suffers due to excessive thrashing, i.e., loading and spilling pages over and over. In the remainder of our experiments, DuckDB uses the Mixed eviction policy as a decent compromise between PersistentFirst and TemporaryFirst.

Allocation Performance. Allocation latencies can significantly affect query performance [21]; therefore, we perform a micro-benchmark to investigate how the proposed buffer manager affects allocations, similar to what was done for Cooperative Memory Management [13]. We measure the time it takes to allocate a small region of $262{,}144$ bytes, which is DuckDB's fixed page size, and the time it takes to allocate a large region of $268{,}435{,}456$ bytes, which is $1{,}024$ times larger. Timings are averaged over $1{,}024$ allocations. Simply allocating this using `jemalloc` [35], DuckDB's internal allocator, takes 1.5 microseconds for the small region and 1.7 microseconds for the large region. When routing these allocations through DuckDB's buffer manager when ample memory is available, these allocations respectively take 1.7 microseconds and 2.0 microseconds. The overhead is negligible and can be explained by bookkeeping, such as incrementing the atomic counter for the current memory usage.

When we fill up memory by loading persistent data, performing these allocations will cause pages to be evicted. Performing the small allocation takes even less time in this scenario: 0.9 microseconds. Only one persistent page needs to be evicted, which is free because persistent pages are replicated in storage, and the allocation is immediately reused.

Performing the large allocation, however, now takes 0.9 *milli*seconds because it causes $1{,}024$ persistent pages to be evicted. Unlike the previous scenario, the pages cannot be reused and must, therefore, be deallocated. Deallocations cause significant overhead, as was also found to be the case for Cooperative Memory Management [13]. If a page size of 4 KiB was used rather than DuckDB's default page size of 256 KiB, the large allocation would have caused $65{,}536$ deallocations instead, causing even more overhead.

Given that the deallocation overhead only occurs when memory is full and only for the variable-size allocations that are very sparingly used in DuckDB, we find this overhead acceptable.

VIII. EVALUATION

In this section, we experimentally evaluate our implementation and compare it with other implementations. We compare our system, DuckDB [20] (version 0.9.2), an in-process open-source DBMS for OLAP workloads that has a vectorized query execution engine, against three other systems with strong aggregation performance:

ClickHouse [36] (version 23.11.1), an open-source column-oriented DBMS for OLAP workloads.

HyPer [11] (version 2023.3), a main-memory-based relational DBMS for mixed OLTP and OLAP workloads, which uses data-centric code generation, developed at Technische Universität München (TUM), now Tableau's data engine.

Umbra [9] (v0.1 2023-11-21), HyPer's successor, a disk-based DBMS with in-memory performance, which also uses data-centric code generation, also developed at TUM.

We do not compare with traditional database systems such as PostgreSQL or MySQL as these are orders of magnitude slower than modern OLAP systems at aggregation for various reasons, such as large amounts of per-tuple overhead [10].

We have verified that the aggregation query described in Section VI leads to the same query execution plan in all systems in the benchmark. We run each query five times and report the median execution time. If queries do not finish within 10 minutes, they are timed out.

Thin Groupings. Table II shows the results for the thin variant of all groupings. We show only scale factors 2, 8, 32, and 128, as a wider table does not fit. We summarize the

TABLE III

EXECUTION TIME IN SECONDS FOR THE WIDE VARIANT OF ALL GROUPINGS AT SCALE FACTORS 2, 8, 32, AND 128. LOWER IS BETTER. THE LOWEST EXECUTION TIMES ARE HIGHLIGHTED IN BOLD. 'A' DENOTES THAT THE QUERY WAS ABORTED. 'T' DENOTES THAT THE QUERY TIMED OUT AFTER 600 SECONDS. WE SUMMARIZE BY NORMALIZING EXECUTION TIMES TO DUCKDB AND THEN TAKING THE GEOMETRIC MEAN.

SF	2				8				32				128			
System	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um
Grouping 1	0.04	0.19	0.03	**0.02**	0.16	0.67	0.10	**0.06**	0.63	2.57	**0.38**	A	2.57	9.91	**1.52**	A
2	0.42	0.34	0.23	**0.22**	2.25	1.45	1.07	**0.77**	52.79	**18.41**	211.11	A	347.28	**111.66**	499.04	A
3	0.43	0.59	0.27	**0.23**	2.30	2.56	1.25	**0.91**	30.00	**26.39**	243.30	A	256.29	**133.50**	T	A
4	0.53	0.51	0.27	**0.23**	2.77	2.43	1.28	**0.91**	53.46	**26.19**	255.55	A	350.96	**122.97**	T	A
5	0.25	0.58	0.23	**0.13**	0.80	2.96	0.89	**0.44**	4.63	30.16	**3.13**	A	**67.56**	A	287.87	A
6	0.23	0.72	0.29	**0.17**	1.01	3.24	1.22	**0.67**	4.96	33.20	**4.56**	A	**72.69**	150.75	378.23	A
7	0.51	0.75	0.35	**0.29**	2.56	3.34	1.76	**1.22**	**30.61**	31.85	382.80	A	**260.12**	A	T	A
8	0.87	1.01	0.42	**0.36**	3.94	4.57	1.96	**1.47**	68.49	**38.79**	487.08	A	**407.26**	A	T	A
9	0.59	0.88	0.47	**0.32**	3.22	4.24	2.18	**1.41**	46.12	**36.37**	451.81	A	**331.08**	A	T	T
10	0.75	1.00	0.61	**0.42**	3.69	4.91	2.69	**1.74**	64.84	**43.70**	585.54	A	**399.46**	A	T	A
11	0.64	0.83	0.61	**0.38**	3.22	3.75	2.71	**1.60**	60.04	**36.41**	530.67	A	**396.53**	A	T	A
12	0.62	0.79	0.62	**0.38**	3.32	3.76	2.70	**1.58**	60.73	**36.00**	533.54	A	**382.09**	A	T	A
13	0.66	0.83	0.60	**0.38**	3.20	3.85	2.67	**1.59**	54.55	**37.00**	534.56	A	**359.95**	A	T	A
Geometric Mean Normalized to DuckDB	1.00	1.53	0.77	**0.54**	1.00	1.45	0.70	**0.45**	**1.00**	1.06	4.54	A	**1.00**	A	T	A

execution times per scale factor by first normalizing to, i.e., dividing by, DuckDB's execution time and then taking the geometric mean, as this weighs each query fairly [37].

For scale factors 2, 8, and 32, the data fits entirely in memory, and the execution times of the systems, as well as the normalized geometric means, are similar, with DuckDB and Umbra being the fastest. One exception is HyPer for grouping 10 at scale factor 32, which takes an order of magnitude longer than the other three systems. For this grouping, HyPer has already switched to its external aggregate implementation. We observed HyPer's external aggregate to use a minimal amount of memory, less than 2 GB out of the available 32 GB.

For scale factor 128, some of the larger groupings do not fit in memory anymore. Note that the exact point where this happens differs per system. For almost all groupings at this scale factor, HyPer uses its external aggregation, which is much slower than its in-memory aggregation. Therefore, HyPer can barely finish some of the queries within the 600-second timeout, resulting in a much higher normalized geometric mean. Umbra does not yet have an external aggregation implementation and has to abort queries at this scale factor. Both DuckDB and ClickHouse are able to finish all groupings well within the 600-second timeout.

We now look closer at the scaling behavior for specific thin groupings. In Figure 5, we show the execution times for the thin variants of groupings 3, 6, and 13 at scale factors 1 through 128. Note that both the x- and y-axis are on a logarithmic scale. The data fits in memory up to scale factor 32, and all four systems show linear scaling, as expected from hash aggregation. HyPer switches to external aggregation for all three groupings at scale factor 128, and, surprisingly, despite yielding a smaller result size than grouping 6 and 13, HyPer uses external aggregation for grouping 3 at scale factor 64. This switch results in a significant performance cliff, as execution time increases by more than 10x, despite the data size growing by only 2x. For DuckDB and ClickHouse, there is a much less noticeable performance 'bump', as execution times only increase by $\approx$ 3x. Umbra runs out of memory and is unable to complete the queries at scale factors 64 and 128.

This figure shows that switching from an in-memory to an external algorithm is not robust as it causes performance cliffs: large and sometimes unpredictable spikes in performance. DuckDB's hash aggregation is much more robust.

Wide Groupings. Table III shows the results for the wide variant of all groupings at scale factors 2, 8, 32, and 128. Compared to Table II, there is a clear difference, as all execution times are higher due to having to scan and materialize the additionally selected columns. Like before, the execution times of the four systems are similar at the lower scale factors, 2 and 8, with Umbra being the clear winner, as evidenced by the execution times and normalized geometric mean. Out of the four systems here, DuckDB is the only system that does not use just-in-time (JIT) compilation for aggregate functions for portability reasons. JIT compilation is known to significantly speed up aggregation performance [38], which explains why DuckDB's execution times are slightly higher.

At scale factor 32, there are already significant differences in execution time. HyPer switches to its external aggregation for most queries due to the increased memory pressure caused by

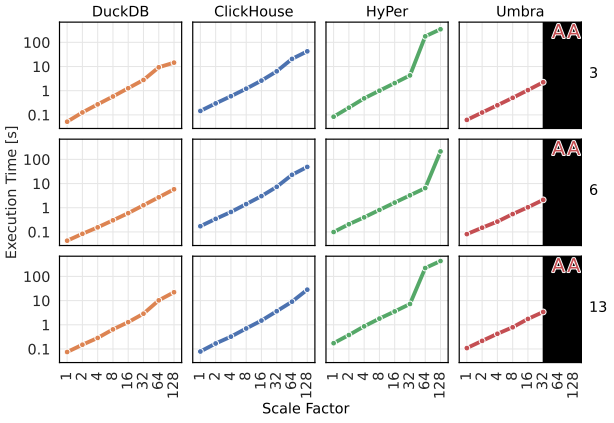

Fig. 5. Execution times for the thin variant of groupings 3, 6, and 13 at scale factors 1 through 128 (log-log scale). Lower is better. 'A' denotes that the query was aborted.

selecting additional columns. As a result, HyPer is slower than ClickHouse and DuckDB by an order of magnitude. Umbra cannot complete any grouping at this scale factor and beyond.

ClickHouse performs well at scale factor 32, having the fastest execution time on most queries and almost the lowest normalized geometric mean if it were not for groupings 5 and 6. However, its strategy to achieve this performance does not hold up, as ClickHouse has to abort most queries at scale factor 128. At scale factor 128, DuckDB is the only system that can complete the benchmark within the 600-second time limit and the 32 GB of main memory available.

We again look at the scaling behavior for the same groupings as before, 3, 6, and 13, but now the wide variants, in Figure 6. All systems show linear behavior for the first few scale factors. However, performance degradation starts earlier, at around scale factor 32, although this depends on the number of rows the grouping yields. HyPer switches to external aggregation for the higher scale factors for all three groupings. It switches at scale factors 32, 64, and 16 for groupings 3, 6, and 13, respectively. The external aggregation sharply degrades performance compared to the in-memory aggregation and finally causes HyPer to time out at scale factor 128 on two out of three groupings. Umbra is not able to complete any of the groupings at scale factor 32 and above.

ClickHouse scales well, especially for the higher scale factors, and although its aggregation strategy can utilize storage to be able to process larger-than-memory intermediates, it still runs out of memory for the largest grouping, grouping 13, at scale factor 128. DuckDB does not have this problem, and again, although DuckDB has a slight performance bump around scale factor 32/64, it can complete all groupings at the highest scale factor without problems, with more than three minutes to spare until the 600-second timeout.

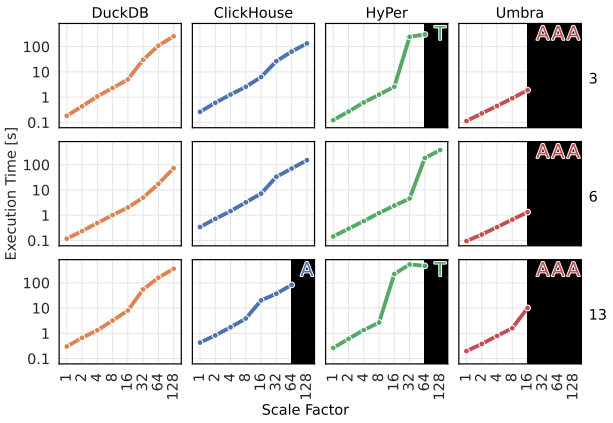

Fig. 6. Execution times for the wide variant of groupings 3, 6, and 13 at scale factors 1 through 128 (log-log scale). Lower is better. 'T' denotes that the query timed out after 600 seconds. 'A' denotes that the query was aborted.

IX. Conclusion & Future Work

In this paper, we have discussed temporary query intermediates, specifically for OLAP systems. We have identified that current approaches for intermediates have limitations that impair memory utilization and efficient spilling. To address these problems, we have proposed Unified Memory Management, which unifies memory management for temporary and persistent data and a page layout that can be lazily spilled without serialization overhead. Together, these techniques allow blocking operators to utilize all available memory for intermediates and to efficiently and lazily spill them. This allows operator implementations that need to process larger-than-memory intermediates to do so without sacrificing in-memory performance.

We have integrated both techniques into DuckDB's parallel hash aggregation. We have experimentally evaluated our implementation and compared it against three systems with strong aggregation performance, using a benchmark with a diverse aggregation workload and varying amounts of unique groups. The results showed that DuckDB can aggregate larger-than-memory intermediates without falling off a performance cliff. This can be attributed to the proposed techniques and the high I/O throughput of modern SSDs. The results also showed that DuckDB's aggregation is highly competitive when intermediates fit in main memory, demonstrating that robust external aggregation performance can indeed be achieved without sacrificing in-memory performance.

Future Work. The proposed techniques were implemented and evaluated for OLAP. More research is needed to determine their viability for OLTP, where smaller page sizes, dirty pages, and many concurrent writers are common.

Our experimental results showed that the efficiency of eviction policies is workload-dependent. Prior research in this area has not considered evicting temporary data. More research into eviction policies for Unified Memory Management is needed to determine efficient strategies.

As mentioned in Section V, the same group may be materialized multiple times during thread-local pre-aggregation, possibly across multiple threads due to morsel-driven parallelism. These additional materializations can cause the size of the temporary query intermediates to grow large, potentially causing unnecessary I/O. This can be mitigated by adaptively exchanging and aggregating partitions early during this first phase if the memory limit would otherwise be exceeded, reducing the size of the intermediates.

Besides hash aggregation, other blocking operators can benefit from the techniques proposed in this paper, such as the join, sort, and window operators. We believe that, by integrating the techniques presented here, these operators can also support the processing of larger-than-memory temporary query intermediates in a performance-robust way.

Finally, when probing two large hash tables in the same pipeline, for example, or performing a high cardinality aggregation over a large join, multiple memory-intensive operators are active simultaneously. In such cases, coordination is required to ensure that the combined memory usage of all active operators does not exceed the memory limit. An approach that adapts to such memory requirements during query execution would be able to assign memory more fairly than a static approach, and, therefore, utilize memory more efficiently.

References

[1] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, no. 2, p. 121–137, jun 1979. [Online]. Available: https://doi.org/10.1145/356770.356776

[2] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Trans. on Knowl. and Data Eng.*, vol. 4, no. 6, p. 509–516, dec 1992. [Online]. Available: https://doi.org/10.1109/69.180602

[3] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation Techniques for Main Memory Database Systems," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 1–8. [Online]. Available: https://doi.org/10.1145/602259.602261

[4] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 54–65. [Online]. Available: https://doi.org/10.5555/645925.671364

[5] D. Lomet, "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed," in *Proceedings of the 14th International Workshop on Data Management on New Hardware*, ser. DAMON '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3211922.3211927

[6] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "LeanStore: In-Memory Data Management beyond Main Memory," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 185–196. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00026

[7] A. Crotty, V. Leis, and A. Pavlo, "Are You Sure You Want to Use MMAP in Your Database Management System," in *CIDR 2022, Conference on Innovative Data Systems Research.*, 2022. [Online]. Available: https://db.cs.cmu.edu/papers/2022/p13-crotty.pdf

[8] V. Leis, A. Alhomssi, T. Ziegler, Y. Loeck, and C. Dietrich, "Virtual-Memory Assisted Buffer Management," *Proc. ACM Manag. Data*, vol. 1, no. 1, may 2023. [Online]. Available: https://doi.org/10.1145/3588687

[9] T. Neumann and M. J. Freitag, "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*. Amsterdam, Netherlands: www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[10] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *Second Biennial Conference on Innovative Data Systems Research, CIDR, Online Proceedings*. Asilomar, CA, USA: www.cidrdb.org, 2005, pp. 225–237. [Online]. Available: http://cidrdb.org/cidr2005/papers/P19.pdf

[11] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, p. 539–550, jun 2011. [Online]. Available: https://doi.org/10.14778/2002938.2002940

[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 *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 743–754. [Online]. Available: http://dl.acm.org/citation.cfm?id=2588555

[13] R. Lasch, T. Legler, N. May, B. Scheirle, and K.-U. Sattler, "Cooperative Memory Management for Table and Temporary Data," in *Proceedings of the 1st Workshop on Simplicity in Management of Data*, ser. SiMoD '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3596225.3596230

[14] G. Graefe, "Volcano— An Extensible and Parallel Query Evaluation System," *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 1, p. 120–135, feb 1994. [Online]. Available: https://doi.org/10.1109/69.273032

[15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How Good Are Query Optimizers, Really?" *Proc. VLDB Endow.*, vol. 9, no. 3, p. 204–215, nov 2015. [Online]. Available: https://doi.org/10.14778/2850583.2850594

[16] G. Graefe, W. Guy, H. A. Kuno, and G. Paullley, "Robust Query Processing (Dagstuhl Seminar 12321)," *Dagstuhl Reports*, vol. 2, no. 8, pp. 1–15, 2012. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3754

[17] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans. Database Syst.*, vol. 11, no. 3, p. 239–264, aug 1986. [Online]. Available: https://doi.org/10.1145/6314.6315

[18] F. M. Schuhknecht, J. Dittrich, and A. Sharma, "RUMA Has It: Rewired User-Space Memory Access is Possible!" *Proc. VLDB Endow.*, vol. 9, no. 10, p. 768–779, jun 2016. [Online]. Available: https://doi.org/10.14778/2977797.2977803

[19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1075–1086. [Online]. Available: https://doi.org/10.1145/1376616.1376723

[20] M. Raasveldt and H. Mühleisen, "DuckDB: An Embeddable Analytical Database," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1981–1984. [Online]. Available: https://doi.org/10.1145/3299869.3320212

[21] D. Durner, V. Leis, and T. Neumann, "On the Impact of Memory Allocation on High-Performance Query Processing," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3329785.3329918

[22] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-Scalar RAM-CPU Cache Compression," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06. USA: IEEE Computer Society, 2006, p. 59. [Online]. Available: https://doi.org/10.1109/ICDE.2006.150

[23] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data Page Layouts for Relational Databases on Deep Memory Hierarchies," *The VLDB Journal*, vol. 11, no. 3, p. 198–215, nov 2002. [Online]. Available: https://doi.org/10.1007/s00778-002-0074-9

[24] M. Zukowski, N. Nes, and P. Boncz, "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing," in *Proceedings of the 4th International Workshop on Data Management on New Hardware*, ser. DaMoN '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 47–54. [Online]. Available: https://doi.org/10.1145/1457150.1457160

[25] L. Kuiper and H. Mühleisen, "These Rows Are Made for Sorting and That's Just What We'll Do," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. New York, NY, USA: IEEE, 2023, pp. 2050–2062. [Online]. Available: https://doi.org/10.1109/ICDE55515.2023.00159

[26] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then, "Get Real: How Benchmarks Fail to Represent the Real World," in *Proceedings of the Workshop on Testing Database Systems*, ser. DBTest'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3209950.3209952

[27] M. Raasveldt and H. Mühleisen, "Don't Hold My Data Hostage: A Case for Client Protocol Redesign," *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1022–1033, Jun. 2017. [Online]. Available: https://doi.org/10.14778/3115404.3115408

[28] W. McKinney, "Introducing Apache Arrow Flight: A Framework for Fast Data Transport," 2019. [Online]. Available: https://web.archive.org/web/20230324080635/https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight

[29] P. Boncz, T. Neumann, and O. Erling, "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark," in *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*. Berlin, Heidelberg: Springer-Verlag, 2013, p. 61–76. [Online]. Available: https://doi.org/10.1007/978-3-319-04936-6_5

[30] T. Do, G. Graefe, and J. Naughton, "Efficient Sorting, Duplicate Removal, Grouping, and Aggregation," *ACM Trans. Database Syst.*, vol. 47, no. 4, jan 2023. [Online]. Available: https://doi.org/10.1145/3568027

[31] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 102–111. [Online]. Available: https://doi.org/10.1145/93597.98720

[32] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang, "DB2 with BLU Acceleration: So Much More than Just a Column Store," *Proc. VLDB Endow.*, vol. 6, no. 11, p. 1080–1091, aug 2013. [Online]. Available: https://doi.org/10.14778/2536222.2536233

[33] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[34] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, jan 2012. [Online]. Available: https://doi.org/10.1145/2071379.2071383

[35] J. Evans, "A scalable concurrent malloc (3) implementation for FreeBSD," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[36] B. Imasheva, A. Nakispekov, A. Sidelkovskaya, and A. Sidelkovskiy, "The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse," in *Optimization of Complex Systems: Theory, Models, Algorithms and Applications, WCGO 2019, World Congress on Global Optimization, Metz, France, 8-10 July, 2019*, ser. Advances in Intelligent Systems and Computing, vol. 991. Springer, 2019, pp. 820–828. [Online]. Available: https://doi.org/10.1007/978-3-030-21803-4_82

[37] P. J. Fleming and J. J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," *Commun. ACM*, vol. 29, no. 3, p. 218–221, mar 1986. [Online]. Available: https://doi.org/10.1145/5666.5673

[38] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask," *Proc. VLDB Endow.*, vol. 11, no. 13, p. 2209–2222, sep 2018. [Online]. Available: https://doi.org/10.14778/3275366.3284966