Saving Private Hash Join

Laurens Kuiper
CWI
Amsterdam
laurens.kuiper@cwi.nl

Paul Groß
CWI
Amsterdam
paul.gross@cwi.nl

Peter Boncz
CWI
Amsterdam
peter.boncz@cwi.nl

Hannes Mühleisen
CWI
Amsterdam
hannes.muehleisen@cwi.nl

ABSTRACT

Modern analytical database systems offer high-performance in-memory joins. However, if the build side of a join does not fit in RAM, performance degrades sharply due to switching to traditional external join algorithms such as sort-merge. In streaming query execution, this problem is worsened if multiple joins are evaluated simultaneously, as the database system must decide how to allocate memory to each join, which can greatly affect performance.

We revisit larger-than-memory join processing on modern hardware, aiming for robust performance that avoids a "performance cliff" when memory runs out, even in query plans with many joins.

To achieve this, we propose three techniques. First, an adaptive, external hash join algorithm that stores temporary data in a unified buffer pool that oversees temporary and persistent data. Second, an optimizer that creates expressions to compress columns at runtime, reducing the size of materialized temporary data. Third, a strategy for dynamically managing the memory of concurrent operators during query execution to reduce spilling.

We integrate these techniques into DuckDB and experimentally show that when processing memory-intensive join query plans, our implementation gracefully degrades performance as the space requirement exceeds the memory limit. This greatly increases the size of datasets that can be processed on economical hardware.

PVLDB Reference Format:
Laurens Kuiper, Paul Groß, Peter Boncz, and Hannes Mühleisen. Saving Private Hash Join. PVLDB, 18(8): 2748-2760, 2025.
doi:10.14778/3742728.3742762
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lnkuiper/experiments.

1 INTRODUCTION

Traditional database management systems (DBMS) are optimized for disk access and they implement sort-based query operators that use external sorting [22] for query intermediates because only small amounts of RAM were available. These systems could process workloads larger than RAM, albeit slowly. Modern DBMSes optimize for main memory access [6, 15] and implement hash-based query operators [11] because large amounts of RAM are available. Although their in-memory performance has drastically improved compared to their traditional disk-based counterparts, their larger-than-memory processing performance leaves much to be desired.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742762

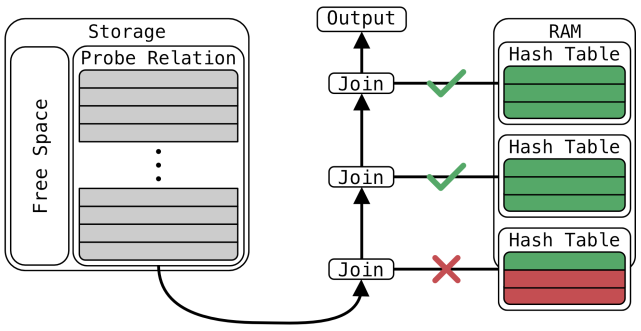

Figure 1: A query plan where three hash joins are probed in a single pipeline. The combined space required for the hash tables cannot exceed RAM; therefore, to "save" the hash joins (instead of using sort-merge), the joins must "spill" some data to storage. Query engines must also decide how to distribute RAM over the joins, which affects the amount of data spilled in each join and the total amount of spilled data.

Shortcomings in external query processing are particularly present in analytical (OLAP) systems, which became mainstream after DBMSes optimized for main memory, despite OLAP systems frequently processing large volumes of data and having large intermediates. Many analytical systems either abort queries because they do not support larger-than-memory intermediates or switch to a traditional disk-based algorithm that is orders of magnitude slower, introducing a "performance cliff" [23]. For large grouped aggregations with many unique keys, this cliff can be avoided using adaptive algorithms [12, 23].

For *joins*, merely implementing an adaptive algorithm is *not enough* to enable robust external query processing. If a single memory-intensive operator, like grouped aggregation, is evaluated in a query plan, all memory is available to that operator. The *join*, however, has two inputs. One of the join's inputs is materialized and could require considerable memory, while the other can usually be streamed to the next operator in the query plan. Therefore, multiple memory-intensive operators can be active simultaneously. We show such a query plan in Figure 1. The combined memory usage of active operators cannot exceed the memory limit, and the available RAM must be distributed. Some distributions may under-utilize RAM and over-utilize storage; therefore, efficient distribution is critical to query performance [1, 8, 32].

Efficient utilization of secondary storage is key to providing good performance at a low cost [29], especially given the high bandwidth of solid-state drives available today. If OLAP systems implemented robust external query processing, i.e., adaptive algorithms and memory control of multiple active operators, they could perform analytical workloads on much more economical hardware. This would then reduce the cost of challenging analytical workloads such as large self-joins, join plans including Parquet files, which are difficult to reorder because only lightweight statistics [30] are available, and join queries with strings, which are ubiquitous in real-world data [44] and require more memory than integer keys.

Contributions. This paper revisits *external joins*, *compressed execution*, and *multi-operator memory control* for modern OLAP systems to enable robust larger-than-memory analytical join processing. Our contributions are the following:

① A morsel-driven [27] *parallel* and *external* **hash join algorithm** that integrates *unified memory management* and *a spillable page layout for intermediates* [23], which gracefully degrades performance as the memory limit is exceeded.

② An optimizer that uses statistics to create expressions to **compress** and **decompress** columns *during execution*, reducing the space requirement of materialization without the need to modify the operators.

③ A dynamic **multi-operator memory control** approach that efficiently distributes the available memory over simultaneously active operators, with a *low synchronization overhead* to prevent degradation of parallel query execution performance.

We have implemented these techniques in DuckDB[1] [38], an in-process OLAP DBMS with a vectorized execution engine. All three are available in the v1.2.0 release. DuckDB is *no research prototype* but a widely used and well-tested system.

Outline. The rest of the paper is organized as follows. Section 2 discusses related work. After briefly describing how DuckDB manages temporary data in Section 3, we present our external hash join in Section 4. We describe our approach to compressed execution in Section 5 and managing concurrent operators' memory in Section 6. In Section 7, we describe our experimental setup, which we use to evaluate and compare our implementation with other systems in Sections 8 to 10. Finally, we conclude the paper and discuss future research in Section 11.

2 RELATED WORK

This section discusses related work on larger-than-memory joins, compressed execution, and multi-operator memory control.

Join Algorithms. The accepted approach to external joins in traditional disk-based database systems is sort-merge [3], using an external sort implementation [22]. Once RAM prices decreased, the efficiency of hash-based algorithms over sort-based algorithms was quickly recognized [11]. Hash joins have become the most common way to join relational data. However, a simple hash join cannot offload (spill) data to storage. Not all workloads fit in memory; therefore, disk-based algorithms remain necessary.

In most systems, the query planner must try to choose an efficient join algorithm, often based on cardinality estimates, which is *risky*. Many systems produce significant estimation errors that grow as the number of joins increases [28]. A poor choice has enormous implications, and in some cases, inserting *a single row* into one of the input tables can cause algorithm choice to change. Choosing a simple hash join can cause queries to abort if the hash table does not fit in memory. Choosing sort-merge can cause execution time to increase by orders of magnitude if a hash table would fit.

GRACE [21] eliminates this decision with a hash join algorithm that performs reasonably well in main memory while also being able to process more data than fits in RAM, allowing it to perform an external join while avoiding the $O(n \log n)$ time complexity of sorting. GRACE is also parallelizable: partitions are independent; therefore, they can join in parallel. However, skewed partition sizes create load-balancing issues. Alternatively, each thread can build a hash table on some tuples from every partition of the inner relation and probe it with *all* tuples from the outer relation [21]. This approach resists skew but increases the total probing effort.

Hybrid Hash Join (HHJ). HHJ [40] improves GRACE hash join by keeping the first partition in memory and using it to perform a simple hash join, only performing GRACE hash join for the other partitions. HHJ equals a simple hash join if all data fits in memory and gracefully degrades to GRACE hash join as the data size exceeds the memory limit. Implementing a single algorithm for in-memory and larger-than-memory joins leads to more robustness: the optimizer no longer has to decide between join algorithms with vastly different performance characteristics.

Research on HHJ has focused on making the algorithm more adaptive. Instead of statically partitioning, a dynamic strategy can be used [33] that does not depend on statistics and is, therefore, resistant to skew. Recent research [20] explores dynamic partitioning policies for HHJ, which can reduce spilling and, therefore, I/O cost. The Memory-Contention Responsive Hash Join [9] is a variant of HHJ that adapts to fluctuations in memory contention at runtime, i.e., it can increase or decrease its memory usage during execution.

Very little research has focused on the parallelism of HHJ. Mourad et al. [31] derive a limit on the parallelism of HHJ under skew when each thread processes a partition, but the authors offer no improvements to the algorithm. HHJ can also execute in parallel using *plan-driven* parallelism, which splits query plans into fragments and connects them through the *exchange* [16] operator, keeping operators largely *unaware of parallelism*. However, plan-driven parallelism also suffers from skewed data distributions. *Morsel-driven* parallelism [27] addresses many of the problems of plan-driven parallelism and can achieve near-linear speedups with additional CPU cores but requires operators to be *parallelism-aware*.

Streaming Query Execution. In streaming query execution such as Volcano [17], tuples *stream* through pipelined query operators until a pipeline breaker, i.e., a *blocking* operator. Streaming reduces memory footprint and has a favorable memory access pattern. Modern analytical streaming query execution paradigms, such as *vectorization*, pioneered by VectorWise [7], or *data-centric code-generation*, pioneered by HyPer [34], take advantage of this. 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.

G-join [18] is a sort-based join algorithm that performs well for inputs that fit in memory but can adapt to larger-than-memory inputs in a performance-robust way. However, G-join always fully *materializes* the outer relation, even if the inner relation fits in memory; therefore, more memory bandwidth is needed, which can quickly become a performance bottleneck [6], and much of the benefit of streaming query execution is lost.

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

Radix hash join [43] is a hardware-conscious join algorithm, popular in literature in the last decade, which also materializes both input relations. However, a thorough evaluation of this algorithm [2] revealed that it only performs better than the simple hash join for specific inputs because materializing the tuples dominates the overall runtime, especially for selective joins.

Compressed Execution. The size of data streamed through query operators is low; therefore, compressing it does not meaningfully reduce memory footprint. However, blocking operators have a much higher memory footprint because they materialize data. *Compressed execution* [19] is a technique that compresses integer and string values in hash tables to reduce the memory footprint. This has been shown to improve the performance of in-memory workloads, and it can most likely improve the performance of larger-than-memory workloads by reducing the amount of data that needs to be spilled to storage.

Multi-operator Memory Control. In non-streaming query execution, a single relational operator is active at any given time. In streaming query execution, multiple memory-intensive operators may be active concurrently *within a single query plan* when joins are involved. Active operators must be assigned a portion of the memory pool, as their combined memory usage may not exceed the limit. A naive approach would be to reserve a fixed amount of memory for each query and each operator. However, in many cases, this will lead to over- or under-utilization of memory, namely when the fixed amount is larger or smaller than the required amount. Under-utilization of memory is especially detrimental for the Hybrid Hash Join, as it reduces the effectiveness of the initial in-memory hash join, thereby requiring more tuples from the (often larger) outer relation to be partitioned and spilled.

Dageville et al. [8] implement a *dynamic* approach to multi-operator memory control that adapts to the workload during execution by applying cost-based rules. A global memory manager publishes a memory bound at three-second intervals, to which operators must promptly react. This approach improves throughput and reduces memory footprint compared to reserving a fixed amount per operator or thread. However, it maintains the same memory limit for all operators in a query plan; therefore, a memory-intensive operator cannot exceed this bound, even if the other operators in the query plan are less memory-intensive. In such cases, query performance suffers due to underutilization of RAM.

Another dynamic approach to memory control is proposed in [42]. However, this approach adapts to workloads that change over time; therefore, it does not apply to the problem of managing the memory of concurrently active operators within a single plan.

Aguilar-Saborit et al. [1] remark that dynamic adjustments can be expensive and propose a *static* approach to multi-operator memory control that assigns allocations to operators before execution. Their approach applies to bushy plans and builds on prior work that only considered left-deep query plans [32]. A post-optimization phase identifies operators in the query plan that are active concurrently and enumerates memory assignments to find a near-optimal solution. This approach allows memory-intensive operators to use more memory than other operators and considers that not all operators are active simultaneously; therefore, it can assign more memory to the active operators, improving RAM utilization.

An obvious shortcoming of static approaches is that they assign memory based on cardinality estimates; therefore, they are prone to over- or underutilization of RAM due to estimation errors. Another shortcoming of the dynamic and static approaches discussed so far is that they reserve memory for joins *too early*. Lang et al. [25] delay allocating a hash table and inserting tuples into it until *after* materializing the inner relation, such that building the hash table can be fully parallelized. If temporary data is spillable [23], this approach can delay a hash join from using significant memory until right before probing starts: only a minimal amount is needed for materialization. This leaves more memory available to other operators, which is especially useful in bushy query plans.

Dynamic memory control does not rely on cardinality estimates to assign memory to operators, as it assigns memory based on true cardinalities observed during execution. Dynamically adapting to workloads causes overhead, especially in parallel query execution. In plan-driven parallelism [16], threads each have their hash join operator; therefore, resizing a hash table does not require cross-thread communication. However, the number of concurrently active operators increases with the number of threads, which *greatly* complicates managing their memory on many-core CPU architectures. In morsel-driven parallelism [27], threads share a single hash join operator, which avoids this problem. However, cross-thread communication is required to resize a shared hash table.

3 MANAGING TEMPORARY DATA

The external hash join that we present in this paper integrates two techniques for managing temporary query intermediates proposed in prior work [23]: ① *unified memory management*, which unifies the memory management of persistent and temporary data, and ② a *page layout* for temporary query intermediates, optimized for in-memory performance, which is spillable to storage without serialization overhead. These techniques are central to our hash join; therefore, we first give an overview.

Unified Memory Management. DuckDB's buffer manager maintains a single memory pool for persistent and temporary data, as proposed by Lasch et al. [26]. Hence, the buffer manager oversees all memory rather than separately maintaining fixed-size pools for persistent and temporary data. This unification allows it to adapt to different workloads better. For example, if the workload is transactional, all available memory can be utilized for persistent data. If needed, all persistent data can be evicted to free up space for a memory-intensive analytical query. In both cases, the available memory is utilized efficiently.

DuckDB's buffer manager not only uses paged allocations for persistent data but also for temporary data. Storing intermediates on pages allows the buffer manager to evict them to storage if intermediates exceed the available memory limit. We implement a specialized page layout for intermediates to avoid serialization when spilling temporary data, which we discuss later in this section.

DuckDB uses a fixed page size of $2^{18} = 262{,}144$ bytes (256 KiB) for all persistent pages. For temporary data, we distinguish three types of allocations. ① *Non-paged allocations* are non-spillable allocations of any size. Despite being non-paged, these allocations go through the buffer manager so that it can keep track of the overall memory usage. ② *Paged fixed-size allocations* have the same size as persistent pages (256 KiB) and, therefore, can be efficiently swapped in and out of a *temporary file* in storage. Using the same size for temporary and persistent pages allows for the reuse of buffer allocations. ③ *Paged variable-size allocations* can also be written to storage, but each page is written to a separate temporary file because it has a variable size.

DuckDB uses variable-size allocations, either paged or non-paged, *only* if efficient query processing requires it, e.g., for the buckets of a hash table or strings larger than DuckDB's fixed paged size, i.e., ≥ 256 KiB. Paged fixed-size allocations are the most common and are used to store almost all temporary query intermediates.

Page Layout. To accommodate storing temporary query intermediates on pages, we have developed a specialized page layout that is *both* efficient for in-memory processing and spillable to storage without any serialization overhead. DuckDB's page layout uses a row-major data representation with fixed-size rows, which is optimal for join/aggregate hash tables [45] and sorting [24]. A row-major layout improves the locality of accessing subsequent attributes in the same row compared to a column-major layout. Their fixed size allows efficient access to the attributes using offsets.

Fixing the row size necessitates storing variable-size data, such as strings, on separate pages and referring to them with pointers. Pointers create a problem of invalid references: if pages storing variable-size data are evicted and loaded back into memory, their addresses may change, invalidating the pointers referencing this data. Therefore, DuckDB's page layout stores a small amount of in-memory metadata alongside the paged data, which describes the layout of the fixed-size rows and variable-size data on the pages.

The metadata stores the page identifiers and the *last-known* addresses of the pages that store variable-size data. When pinning these pages, DuckDB detects if these pages were evicted and loaded into a different address by comparing the last-known and *current* pointers. If the current page pointer *is equal* to the stored page pointer, the buffer manager did not evict the page in between page accesses, and all of the pointers pointing to this page are still *valid*. If the current page pointer *is not equal* to the stored page pointer, all pointers pointing to this page have become *invalid*.

DuckDB then recomputes these pointers by subtracting the last-known page pointer from them, yielding an offset into the page. We then add this offset to the current page pointer to obtain a valid pointer again. Pointer recomputation is inexpensive and happens in place and *lazily*, i.e., only after we have detected that the page that stores variable-size date has gone to disk instead of *pre-emptively*.

Buffer Eviction Policy. After DuckDB materializes temporary data on pages, the pages are unpinned. Unpinning effectively frees up memory, as the buffer manager can offload these pages to storage whenever necessary, allowing for larger-than-memory data materialization. In DuckDB's partitioned external hash aggregation [23], this approach allows for fine-grained spilling of pages instead of coarse-grained approaches that spill entire hash partitions in an all-or-nothing manner. However, a drawback of this approach is that operators no longer have control over which partitions are in memory unless they are pinned. If the buffer manager arbitrarily offloads pages, it may offload pages needed *soon*, causing them to be offloaded and then loaded again unnecessarily. To prevent *thrashing*, the buffer manager should prioritize offloading pages that will be pinned *late*.

To achieve this, we exploit DuckDB's tendency to process hash partitions sequentially, i.e., partition 0, 1, etc. We add partitioning information to pages, which the buffer manager uses to prioritize spilling pages from, e.g., partition 42 before spilling pages from partition 0. Pages from the same partitions are evicted using a *least-recently-used* policy. Persistent data is always evicted before any temporary data, as evicting persistent data does not require writing to storage because it is already stored in the database file. No distinction is made between temporary pages belonging to queries from different connections; they are added to the same queue.

By integrating these techniques for managing temporary data into DuckDB's hash join, it can effortlessly spill data; therefore, it can robustly process larger-than-memory query intermediates by partitioning the data, with only minor modifications to the in-memory algorithm, as will be explained in detail the next section.

4 EXTERNAL HASH JOIN

This section first discusses the considerations that went into designing DuckDB's hash join before presenting the implementation.

Considerations. The external processing capabilities of the hash join should not compromise in-memory performance, and performance should degrade gracefully as the size of the temporary intermediates exceeds the memory limit, like with DuckDB's external hash aggregation [23]. This requirement effectively rules out unnecessarily materializing probe-side data, as this significantly degrades in-memory performance [2] and increases the space requirement. Not compromising on in-memory performance also necessitates deferring the decision to perform any external processing to query execution, as this avoids the optimizer erroneously deciding to deviate from the in-memory strategy based on statistics.

To achieve graceful degradation as the memory limit is exceeded, the join operator should not have a "*hard switch*" when it deviates from the in-memory strategy at execution time, as this will cause a "*performance cliff*". Therefore, the join should continue with the in-memory strategy as much as possible, even if the build side of the join exceeds the memory limit. The extent to which the overall strategy deviates from the in-memory strategy should be proportional to the extent to which the size of the temporary query intermediates exceeds the memory limit, i.e., a different strategy should only be used for the portion of data that does not fit in memory. HHJ [40] achieves this by performing a simple hash join on as much data as will fit in memory and only performing a partitioned hash join on an amount of the data that exceeds the memory limit.

The implementation should also be fully parallel at every step, as any single-threaded execution limits many-core scalability, as per Amdahl's law. The parallelism should be skew-resistant, which rules out parallelization over hash partitions, as data distributions can skew their sizes, causing some threads to perform much more work than others, like with plan-driven parallelism [16]. Instead, we use the concept of morsel-driven parallelism [27] throughout the entire join, always scheduling tasks over *fine-grained* fragments of input data. Although I/O is likely to be the bottleneck when processing query intermediates larger than RAM, modern storage devices are highly concurrent and should be accessed concurrently whenever possible to achieve maximum throughput.

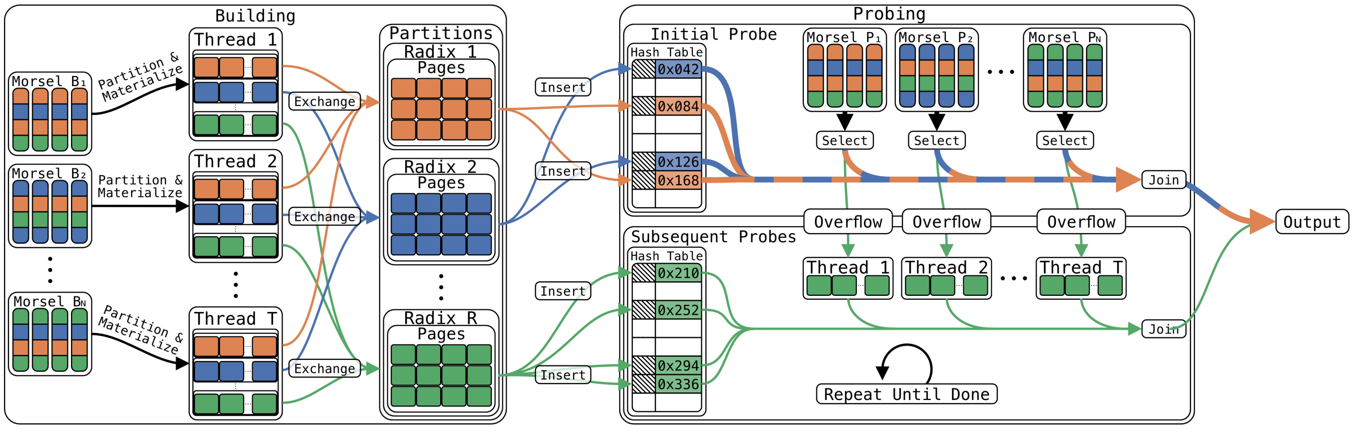

Figure 2: DuckDB's hash join. *Building* is done by assigning morsels to threads until all build-side data has been read. Each thread radix partitions and materializes the data to DuckDB's spillable page layout, and a hash table is built on one or more partitions. Then, *probing* starts with an *initial probe*. Morsels are assigned to threads until all probe-side data has been read. Each thread selects the tuples from the data that can join with the partitions in the hash table and probes and immediately outputs matching tuples. The *overflow* tuples that are not selected are also partitioned and materialized. After the initial probe, all remaining data from both sides has materialized within the join operator, which is processed in a series of *subsequent probes*.

We will now give an overview of DuckDB's hash join implementation, which was designed to fulfill these requirements. DuckDB's in-memory hash join is inspired by HyPer's hash join [27] but with key optimizations and larger-than-memory processing capabilities. Our design is illustrated in Figure 2. The algorithm has multiple distinct phases, which we will explain in detail.

Building. In this phase, we assign morsels to threads until all build-side data has been read. There can be many more morsels than threads. Data is scanned from morsels in batches of up to 2,048 tuples, which is DuckDB's standard vector size. The threads hash the join key columns and directly materialize tuples into partitions that use DuckDB's spillable page layout. As explained in Section 3, the pages are then unpinned; therefore, they can be evicted by the buffer manager when necessary.

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. By materializing and partitioning tuples simultaneously, rather than consecutively, we avoid copying tuples *more than once*, reducing the partitioning cost. This is an important detail because maintaining the performance of the in-memory hash join, which does not strictly require partitioning, unlike the external hash join, is one of our design goals.

Partitions are determined by radix, i.e., a few of the middle bits of the hash. The lower and upper bits of the hash are used to determine the entry in the hash table and improve collision resolution performance, as will be explained. Any of the used bits must not overlap, as this makes them less effective. Initially, only four radix bits are used, creating $2^4 = 16$ partitions. After all build-side data has been partitioned, all thread-local data is exchanged to a global state. If any single partition is too large to fit in memory, the data is repartitioned using more radix bits, creating more partitions.

Hash Table Creation. After the building phase, tuples can be inserted into the hash table. We select as many partitions as will fit in memory to insert into the hash table to maximize memory utilization. If all partitions fit, the join will proceed fully in memory. Checking whether all partitions fit is a simple and inexpensive operation that does not meaningfully affect in-memory performance.

After selecting the partitions, we allocate and zero-initialize an array of 64-bit entries with a large enough capacity to fit an entry for each tuple in the selected partitions. Then, each thread is assigned morsels from the selected partitions to insert into the array until all tuples have been inserted. The tuples' hashes are scanned, and the offsets into this array are obtained from the lower bits of the hash. The pointers to the tuples are added to the 64-bit entries at these offsets using atomic compare-and-swap operations [25].

Collision Resolution. Hash collisions, i.e., tuples hashing to the same bucket, are resolved using a *combination of linear probing and chaining*. We reduce the random access incurred by linear probing using the upper 16 bits of the hash, which we call *salt*. Pointers have a width of 64 bits on 64-bit CPU architectures, but only the lower 48 bits are used for the address, as this already allows for up to $\approx$ 281 TB of address space. We store the salt in the remaining 16 bits of the 64-bit entries. When probing, we first compare the salt. If the salt is equal, we follow the pointer to compare the join keys, like in DuckDB's hash aggregation [23].

However, a key difference between the join and aggregate hash tables is that the join hash table may contain tuples with the *exact same* join keys, which are deduplicated in grouped aggregation. DuckDB's join hash table treats these collisions differently from hash collisions that have *different* join keys.

If we encounter a hash collision (and matching salt) while inserting a tuple into the hash table, we compare its join keys with the join keys of the colliding tuple. If the join keys are not equal, the collision is simply resolved using linear probing. If the join keys are equal, we instead create a *chain*, i.e., a linked list of tuples with the exact same join keys. This enables us to create chains that only feature one unique set of join keys.

By performing these comparisons *once* while building the hash table, we do not have to perform them while probing (possibly *multiple times*). If a probing tuple's join keys are equal to those of the first build tuple in a chain, the probe tuple can be joined with all build tuples in the chain *without further comparisons*. This approach reduces the number of performed comparisons and linear probing collisions in many-to-many joins. Furthermore, these chains could also be used for factorization [14], which is crucial for graph query processing. This optimization also allows us to detect the opposite case, which is when there are *no duplicate join keys at all*. We use this information to improve performance while probing, as we know that there are no hash collision chains; therefore, we can skip (attempting to) follow the chains. Probing starts after the hash table is created. We distinguish two probe phases: the *initial* probe and one or more *subsequent partitioned* probes.

Initial Probe. In this phase, we assign morsels to threads until all probe-side data has been read. The join key columns are hashed for every batch of data the threads receive, and the radix is extracted from the hash. Tuples with a radix that matches one of the inserted build-side partitions can probe the hash table immediately. Hence, these tuples are *selected*. This selection does not imply partitioning or materialization; only the creation of a *selection-vector* [7]. The selected tuples probe the hash table and continue pipelined query execution. Note that all build-side partitions are inserted into the hash table for the in-memory hash join. By default, all probe-side tuples are selected, and their radix does not need to be extracted; therefore, in-memory performance is unaffected.

Some probe-side tuples, those with a radix that does not match one of the inserted build-side partitions, are not selected for external hash joins. These tuples are partitioned and materialized to a spillable page layout. The page layout for the probe-side data is similar to that of the materialized build-side data, as it also uses lazy pointer recomputation [23]. An important difference is that the probe-side data will not be randomly accessed but sequentially scanned; therefore, we do not need a row-major data representation to improve the efficiency of attribute access [45]. Instead, the partitioned probe-side data is stored using a column-major data representation. This is the default for DuckDB's execution engine, allowing the data to be scanned without copying it.

Subsequent Partitioned Probes. For external hash joins, the initial probe is followed by one or more partitioned probes. First, however, after the initial probe, the hash table may have to be scanned, depending on the *join type*; for example, unmatched tuples are scanned if it is a *right* join. The hash table is scanned in parallel by splitting the data into morsels of, e.g., $\approx$ 100,000 tuples. If all build-side data fits into memory, the join is done after completing this scan. If not, the remaining build- and probe-side data that has been partitioned and materialized still needs to be joined and output by performing one or more rounds of partitioned probes.

Each subsequent partitioned probe cycles through three "stages": ① Inserting build-side tuples into the hash table, ② Probing the hash table, and ③ Scanning the hash table (if the join type requires it). A thread-global state oversees the progress and assigns tasks to the active threads. During this phase, the hash table is again built on as many partitions as possible, allowing for longer uninterrupted probing, thereby reducing the need for synchronization. After all partitions have been processed, the join is done.

DuckDB's external hash join has been part of every DuckDB release since v0.5.0. Some of the described optimizations have been added in later releases. With this design, all decisions regarding the external hash join are made dynamically during execution without relying on statistics. In every phase of the join, we have been careful not to compromise the in-memory performance with the larger-than-memory functionality. If the build side exceeds the memory limit, memory utilization is maximized by performing an in-memory hash join on a portion of data that fits in RAM. Although there are some points where threads must synchronize, e.g., to decide which partitions will be inserted in the hash table, the bulk of the work is fully parallelized using morsel-driven parallelism.

In this section, we described how DuckDB's join adheres to a memory limit, but *multiple operators could be active simultaneously*. In Section 6, we present our approach to multi-operator memory control and explain how it is integrated with the hash join.

5 COMPRESSED MATERIALIZATION

External joins require materializing data from both relations. If a quarter of the inner relation fits in RAM, three-quarters of the outer relation must be materialized. Most joins are *asymmetric*, with the inner relation often much smaller than the outer relation. The benefit of fitting, e.g., 10% more of the inner relation in memory is enormous, as it avoids materializing another 10% of the potentially much bigger outer relation.

DuckDB has had a so-called *Compressed Materialization* optimizer since version 0.9.0. This optimizer creates projections to compress columns before materializing operators such as joins and projections to decompress the columns afterward. By implementing compressed execution as an optimizer, instead of within hash tables [19], the (de-)compression logic is centralized and applicable to all operators without modification (operator-agnostic).

Integer Compression. If min/max statistics are available, they are used to apply frame-of-reference compression to compress a *wide* integer type to a *thinner* integer type. If the *range* (max - min) of an integer column is small enough to fit in a thinner type, we subtract the minimum and cast to the thinner type, reducing its width by at least half. This transformation can be reversed by casting back to the wider type and adding the minimum again. It can be applied to *all* columns from the inner relation, even if the column is used as a join key, because the transformation does not affect comparisons as long as the corresponding column from the outer relation is transformed in the same way.

String Compression. DuckDB uses the string type proposed by Umbra [35]. With this representation, the width of an arbitrarily sized `VARCHAR` is 16 bytes, which includes a pointer if the string is longer than 12 characters. If the maximum string length is known to be, e.g., 3 bytes, we store the string in a 32-bit integer instead, saving 12 bytes per value. When a string is converted to an integer type, its comparison properties can be preserved by copying the bytes in reverse order on little-endian machines. This conversion reduces memory usage and speeds up comparisons, as integer comparisons are much more efficient than string comparisons.

These projections are inexpensive, and adding them improves performance substantially because they reduce, for example, hash table memory requirements.

6 MULTI-OPERATOR MEMORY CONTROL

To get a better understanding of how assigning memory to operators affects query performance, we consider an example similar to Figure 1, where an outer relation O joins two inner relations, I_1 and I_2, in a single pipeline, where I_1 is joined first, and then I_2. The pipeline ends in the consumption of the data in operator P, for example, an ungrouped aggregation. We denote the memory limit with L. To simplify our example, we assume that: ① Scanning O and executing P requires no memory, ② O is much larger than L, and its join keys are uniformly distributed, ③ Both joins are non-selective one-to-many relationships, i.e., all tuples from O will find exactly one match in both joins. These assumptions allow us to analyze simplified cases where external processing is required.

Memory Assignments. Consider case Ⓐ where L = 1 GB, and I_1 and I_2 are 1 GB each. We cannot fully perform both joins in main memory; therefore, we must decide how to divide the memory between them. If we assign the first join with I_1 1 GB and the second join with I_2 nothing, the first join can proceed fully in memory, but the second join does not have any memory to build a hash table; therefore, it cannot perform a streaming probe with *any* data coming from O. With this assignment, all of O is fully materialized in the second join (with I_2). According to our assumption, O is much larger than L; therefore, spilling it has a considerable I/O cost and storage space requirement.

If we assign both joins 500 MB of memory, both can perform a streaming probe with half of the probe-side data while having to materialize the other half. With the previous 1,000/0 assignment, all probe-side data is materialized once. The 500/500 assignment seems to have the same implications because half the probe-side data is materialized twice, once in each join. *However*, with the 1,000/0 assignment, the space requirement is twice as large, as the probe-side data is materialized *all at once* in the second join. The 500/500 assignment is preferable because it materializes only up to half of the probe-side data at any given time.

Relation Sizes. The size of the inner relations significantly affects how memory should be assigned. Consider again the same example with L = 1 GB, but now, for case Ⓑ, the inner relations have different sizes: I_1 is 200 MB, and I_2 is 1,800 MB. Choosing for *equality*, i.e., a 500/500 memory assignment, is wasteful as I_1 only needs 200 MB. Choosing for *equity*, i.e., a 100/900 assignment, would again cause half the data to be materialized in both joins. A better assignment would be 200/800, as this allows the first join to proceed fully in memory while only requiring 10/18th of the probe-side data to be materialized in the second join. So, by materializing just 1/18th more of the probe-side data in the second join, we avoid materializing half of O in the first join.

From this example, it seems like more memory should always be given to smaller joins, but this leads to *starvation* of larger joins. Consider case Ⓒ, where I_1 is 1 GB and I_2 is 2 GB. We could assign 1 GB to the first join and none to the second to reduce the total amount of materialized data. With this assignment, all data is materialized in the second join, and none of the data is fully streamed through the pipeline into the consuming operator P; therefore, this assignment increases the overall space requirement. To address this complex trade-off, we propose a cost model that can be optimized dynamically during query execution.

Cost Model. By analyzing the examples in this section, we have identified two aspects to consider when assigning memory to joins: *materialization cost*, which should be minimized, and *pipeline throughput*, which should be maximized. As we have seen, these aspects are at odds, as reducing overall materialization may cause throughput to drop to zero, and increasing throughput may increase materialization cost. Therefore, we have created a cost model that considers materialization cost *and* pipeline throughput.

We express a join's *materialization cost* as the fraction of the probe-side data that must be materialized. This fraction depends on the size s_i of inner relation I_i and how much memory a_i is assigned to the join. We weigh this fraction by a weight w_i, as each join's materialization cost can differ. In DuckDB, w_i is equal to the width of the probe-side tuples, as the size of the materialized data, and, therefore, the cost increases linearly with this width.

We define the total materialization cost $\mathbf{M}(A, S)$ of a join pipeline that joins an outer relation with N inner relations as the sum of these weighted fractions, where $S = \{s_1, s_2, ..., s_N\}$ and $A = \{a_1, a_2, ..., a_N\}$. We express a join's *throughput* as the inverse of the materialization cost: the fraction of the probe-side data that can be streamed through it. This fraction also depends on the operator's size and memory assignment. We define the throughput $\mathbf{T}(S, A)$ of a join pipeline as the geometric mean of these fractions, creating a value between 0 and 1.

We formally define $\mathbf{M}(A, S)$ and $\mathbf{T}(S, A)$ as

$$\mathbf{M}(S, A) = \sum_{i=1}^{N} w_i \cdot \left(1 - \frac{a_i}{s_i}\right), \qquad \mathbf{T}(S, A) = \left(\prod_{i=1}^{N} \frac{a_i}{s_i}\right)^{\frac{1}{N}}.$$

As mentioned, $\mathbf{M}(S, A)$ must be minimized, whereas $\mathbf{T}(S, A)$ must be maximized. We incorporate both into a single cost model $\mathbf{C}(S, A)$ by multiplying the materialization cost with (1 - throughput), i.e., $\mathbf{C}(S, A) = \mathbf{M}(S, A) \cdot (1 - \mathbf{T}(S, A))$.

According to this cost model, the best memory assignments for the cases are: Ⓐ 500/500, Ⓑ 200/800, Ⓒ $666.\bar{6}/333.\bar{3}$, assuming widths w_i are equal for each join. As we can see, the cost model assigns memory equally when the relations are of equal size and prioritizes smaller joins when they are not. It does so without *starving larger joins completely*, as starving any join causes $\mathbf{T}(S, A)$ to quickly become 0, significantly increasing the overall cost due to the multiplication with $1 - \mathbf{T}(S, A)$ in the cost model.

The model is sufficiently complex to express the combination of the materialization cost and pipeline throughput into a single number. It uses materialization and throughput only using relative, not absolute, sizes; therefore, the cost is the same if constant factors scale the assignments, sizes, and memory limit. The inputs to the proposed cost model are *observed values*, not estimated ones, i.e., accurate memory usage is obtained during execution. The model is also not unnecessarily complex; for example, it does not attempt to account for variables such as estimated selectivity or memory and storage bandwidths. Such variables are not guaranteed to remain constant over time; therefore, if inaccurate, they could negatively impact the cost model.

Static vs. Dynamic. As discussed in Section 2, static approaches to multi-operator memory control [1, 32] rely on statistics, which are not always present. Even when present, statistics are prone to producing unreliable cardinality estimates for join order optimization [28], which can produce plans with much larger intermediates and are, therefore, orders of magnitude worse than optimal. If relied upon for multi-operator memory control, these cardinality estimation errors could cause memory to be assigned inefficiently, spilling more data and significantly degrading performance.

We opt for a fully *dynamic* approach where memory control is interleaved with query execution, observing the actual cardinalities in the pipelines as they occur and reacting accordingly. However, we avoid limiting operators' memory usage like the dynamic approach proposed by Dageville et al. [8], as this can cause memory to be underutilized, as discussed in Section 2. Instead, we allow large joins to use more memory, if necessary, like the static approach of Aguilar-Saborit et al. [1].

Dynamically adjusting memory while executing a query can be expensive because hash tables may have to be *resized*. This is especially costly if systems have a high degree of execution parallelism, as threads may have to wait while the hash table is being resized. We must be careful not to impair parallel performance, especially in the typical case where all operators fit in RAM and extensive control of the memory of multiple operators is not needed. Our approach must, therefore, have a *low synchronization overhead*.

Dageville et al. [8] use a background thread to broadcast a memory assignment to each operator at a fixed interval, to which operators must react. Their system uses plan-driven parallelism [16], in which operator parallelism is *encapsulated*. In this paradigm, each thread executes its own plan fragment and has its own copy of each operator. This complicates dynamic multi-operator memory control with high degrees of parallelism, as there are many active operators whose memory usage can be adjusted and whose combined memory usage must not exceed the memory limit.

DuckDB uses morsel-driven parallelism [27], in which operators are parallelism-aware. In this paradigm, there is only one copy of each operator on which threads work in parallel. There are also *fixed synchronization points*, which make it very clear when operators start and end for all participating threads. This simplifies dynamic multi-operator memory control, as it is much easier to get an overview of the memory usage of operators, pipelines, and query plans as a whole in this paradigm.

Dynamic Assignments. In DuckDB, operators indicate the amount of data they have at these fixed synchronization points and receive a memory assignment in return. For our external hash join, this happens right before a hash table is built from materialized data, i.e., before the initial probe and each subsequent partitioned probe. This significantly reduces the communication between operators and the data structure responsible for assigning memory to operators, which we call the Temporary Memory Manager (TMM).

At first glance, this approach may not seem flexible enough to produce efficient memory assignments at runtime. In our running example, the hash table for the join with I_1 may be built first. Some memory would be assigned to this join without taking the join with I_2 into account. The memory assignment of the join with I_1 can only be reduced at this join's own request. This leaves less memory available for the join with I_2, with no way to adapt.

As explained in Section 4, DuckDB first performs *Build-Side Materialization*, in which all build-side data is materialized using a spillable page layout. Large amounts of data can be materialized without pinning pages in memory because the buffer manager can spill them. Only after all build-side data has been materialized, DuckDB performs *Hash Table Creation*, in which tuples are inserted into a hash table, which finally requires data to be in memory.

We exploit the delay between *Build-Side Materialization* and *Hash Table Creation* to produce efficient memory assignments at runtime. We delay building the hash table of the simultaneously active joins until the data for *all* of these joins has been materialized. In our running example, this entails that we first materialize I_1 and I_2. Then, the sizes of I_1 and I_2 are communicated to the TMM. Finally, both joins request a memory assignment from the TMM and build their hash tables accordingly. By delaying the requests for memory assignments until all relations have materialized, the TMM can fairly divide the memory between the active operators with all the information it needs to optimize the cost model.

This approach ensures that only the joins probed in the same pipeline require memory simultaneously. This reduces the problem of assigning memory to active operators to left-deep plans as in [32], effectively eliminating the need to consider bushy plans as in [1].

Cost Model Optimization. When a memory assignment is finally requested, parameter S, i.e., the sizes of the materialized relations of the cost model $\mathbf{C}(S, A)$, is known. The memory assignments, parameter A, are to be decided by the TMM. We approximate the optimal (according to the cost model) memory assignments with a few iterations of gradient descent. First, we initialize each $a_i \in A$ with a minimum assignment that the operator needs. Then, in each iteration, we compute the gradient of $\mathbf{C}(S, A)$ with respect to each a_i. We increase the a_i with the lowest gradient by a fraction of the free memory. Note, however, that, unlike regular gradient descent, this increase is bounded, as the memory assignment a_i of the operator i should not exceed its total size s_i.

To reduce the potential cost of this optimization, we perform a fixed number of iterations of gradient descent, and only a few times per operator, at the aforementioned fixed synchronization points, e.g., before the initial probe and each subsequent probe in the external hash join. This also only happens when query execution can benefit from it, i.e., if the total size of the relations exceeds the memory limit; therefore, in-memory performance is unaffected. The time spent optimizing the cost model is insignificant compared to the time spent processing the larger-than-memory data.

Concurrency and Ownership. In DuckDB, there is one *shared* TMM for all active connections, meaning that the TMM is also responsible for managing the memory of concurrent queries. If multiple connections are concurrently executing join pipelines, the TMM optimizes the cost model as if the operators from all connections *belong to the same pipeline*. The memory that the TMM assigns to operators is *not owned* by the TMM but by the operators themselves. The TMM only gives a limit back to an operator that requests an assignment, which the operator *promises* to stay under.

The concurrency and ownership design of the TMM in DuckDB is an *implementation decision*. If an individual memory limit per connection must be imposed, one TMM should be used per connection instead of a shared TMM for all active connections. Similarly, if centralized ownership of memory allocations is desired, the TMM could *own* the memory it assigns to operators.

The Temporary Memory Manager has been implemented in DuckDB since v0.10.0 and has since been integrated into the hash join and aggregation operators. The TMM, as described here, has been released since v1.1.0.

7 EXPERIMENTAL SETUP

Our experiments run on AWS EC2 using Ubuntu 24. We run each query 5 times and report the median execution time. If any of the 5 runs does not complete within 1,000 seconds, we report a timeout.

Hardware. We choose the `c6id.4xlarge` AWS instance type for our experiments. This instance has an Intel Xeon 8375C CPU with 8 cores (16 threads) and 32 GB of DDR4 RAM. The instance has access to 1 TB of NVMe Instance Storage, which is physically attached to the host, unlike the usual Elastic Block Storage [39] on EC2. We set the instance's tenancy to `dedicated` so that the entire node is reserved, but we do not use the rest of the node's capacity. This setup eliminates the *noisy neighbor* problem that cloud environments may have; therefore, our results are consistent.

Data Generation. We generate data using different parameter configurations, creating a variety of larger-than-memory joins, allowing us to evaluate how these parameters affect performance.

We consider the following four parameters: ① *Cardinality*, the number of rows. We vary this parameter between 100 million and 1,000 million. ② *Width*, the number of *sets* of projected columns. A set of columns consists of one *tag* column (a single uppercase letter), one *employee* column (of length 15, e.g., `EMPNO424242424242`), and one *comment* column (a sentence consisting of four random lorem ipsum words, with an average length of ±27 characters). We vary this parameter between 0 and 4. We only use string columns because they are ubiquitous in real-world data [44] and more challenging to process than fixed-width columns. ③ *Skew*, the skewness of the join keys. The skewness is determined by sampling from a power law distribution with parameter α. We vary α between 0, for which the distribution is uniform, and 1, for which the distribution is Zipfian, and around 75% of join keys are the same. ④ *Unique Key Count*, the number of unique join keys *from which to sample*. We vary this parameter between 40 million and 200 million.

Isolating Performance. While large join pipelines can easily dominate a query's runtime, *isolating* the performance of internal components of database systems is difficult because we can often only reliably observe end-to-end query runtime. Some systems have built-in query profilers, but these differ from system to system; therefore, their measurements cannot create a fair comparison. The end-to-end runtime includes unwanted and unrelated overheads, such as query planning and transferring the result set through a client protocol. Client protocol serialization is especially costly and can dominate a query's runtime [37]. We must reduce the impact of these overheads to measure the performance of joins reliably.

With sufficiently large workloads, any planning overhead becomes insignificant. To reduce transfer protocol overhead, we add an ungrouped aggregation to the query, reducing the result set size to one row. The aggregation must be cheap so that it does not affect execution time. One of the cheapest aggregate functions would be `COUNT(col)`, but this can be replaced with `COUNT(*)` by the optimizer if `col` does not contain any `NULL` values. This would allow the optimizer to remove that column entirely from the query, greatly reducing the overall memory usage. Instead, we use the `ANY_VALUE(col)` aggregate function, which is a cheap aggregate function that returns an arbitrary value from the input. In theory, this yields the same result as adding a `LIMIT 1` clause to the query. *However*, adding `LIMIT 1` allows most systems to terminate queries early after seeing one row. The `ANY_VALUE(col)` approach causes the systems below to fully evaluate the workloads while also yielding a result set of just one row.

Systems. We compare DuckDB [38] (v1.2.0), against one traditional database system and two systems with strong analytical performance. Traditional systems are often orders of magnitude slower than modern OLAP systems at analytical workloads for various reasons, such as large amounts of per-tuple overhead [7]. However, they offer robust external query processing; therefore, they are interesting to compare against. Besides DuckDB, the systems under evaluation are the following:

PostgreSQL [41] (version 16.3) The popular, open-source, traditional, disk-based relational DBMS for OLTP workloads, initially developed at UC Berkeley.

HyPer [34] (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 [35] (v0.1 2024-04-17), HyPer's successor, a "Disk-Based System with In-Memory Performance" that also uses data-centric code generation, developed at TUM, used by CedarDB [10].

For DuckDB, we use `SET allocator_background_threads=true;` to improve `jemalloc` [13] allocation performance. For HyPer and Umbra, we use default settings. For PostgreSQL, we use similar settings to those used by Leis et al. [28], and set `work_mem` to 1 GB, `shared_buffers` to 4 GB, and `effective_cache_size` to 30 GB. We also set `temp_file_limit` to −1, to allow unlimited spilling.

8 EVALUATION: EXTERNAL JOIN

In this section, we experimentally evaluate and compare our external join implementation with other implementations. We perform an inner join of two tables, evaluating only the external join. Multi-operator memory control will be evaluated in Section 9.

Workload. In our experiments, we change one of the data generation parameters described in Section 7 at a time, allowing us to isolate its effect on external joins. We first establish a *default* parameter configuration as a starting point. The default inner relation has a cardinality of 200M rows, a width of 1 set of columns, no skew (i.e., $\alpha = 0$), and 200M unique keys. With these parameters, the size of the inner relation is roughly equal to the available memory in our experimental setup, although due to internal differences, this size is not equal for all systems. DuckDB can fit ±2/3rd of the inner relation in RAM with this configuration. The default outer relation has a cardinality of 500M rows, a width of 1 set of columns, skewed with $\alpha = 0.5$, and 200M unique keys. We create six experiments by varying the inner and outer relation's cardinality, width, and skew.

When varying the cardinality of the inner relation, we ensure that all tuples will be matched by setting the number of unique keys *equal* to the cardinality of the inner relation. Increasing the cardinality/width of the inner and outer relations stresses systems' ability to join larger volumes of data. Execution time should increase linearly with the cardinality/width of the relations. However, algorithm choice and spilling may introduce non-linearity in systems, which this experiment should reveal.

When varying α of the inner relation, we keep α of the outer relation fixed at 0, and vice versa, to avoid joining two skewed

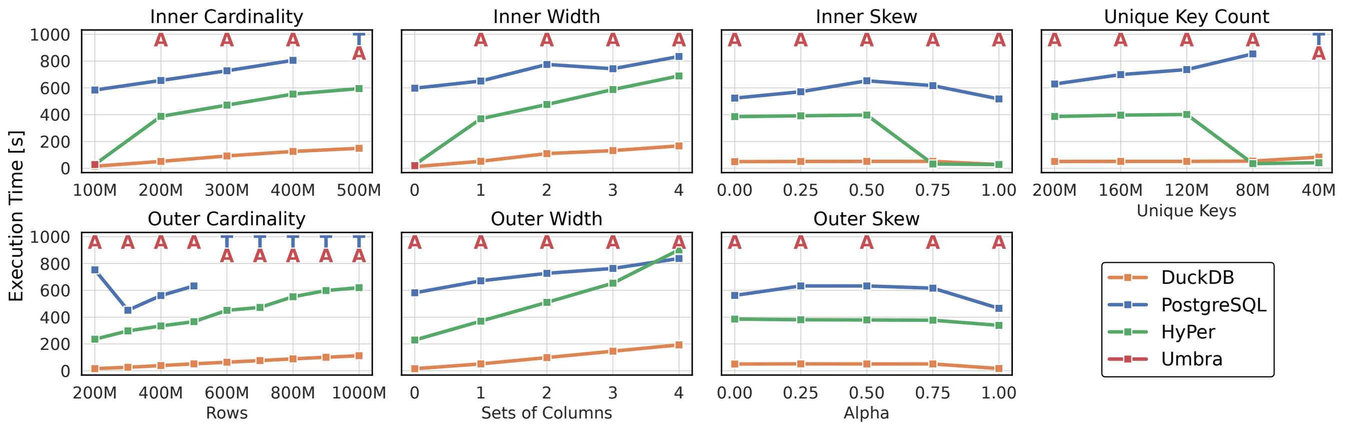

Figure 3: Execution times for external joins with different data generation parameter configurations. Lower is better. 'T' denotes that the query timed out after 1,000 seconds. 'A' denotes that the query was aborted due to running out of memory.

distributions together and almost creating a cross product. For the skew experiments, we expect execution time to decrease slightly for higher α because the random access from probing the hash table will become skewed towards join keys that appear more often, which reduces overall cache misses. For $\alpha = 1$ (Zipfian) data distributions, around 75% of the data belongs to a single partition. These experiments should reveal, e.g., if systems use partitions to parallelize or spill entire partitions in an all-or-nothing manner.

For our final experiment, we vary the unique key count. With lower key counts, tuples can have multiple matches, and the result cardinality is larger than either input relation. For example, with an inner relation cardinality of 200M and a unique key count of 40M, the result cardinality is 5× the cardinality of the outer relation, assuming all tuples find a match. We expect execution time to increase with fewer unique keys, as the join will emit more tuples. This experiment should reveal how well systems deal with *exploding joins*, which may cause issues such as many hash collisions.

Results. We show the results in Figure 3. In the *Inner Cardinality* experiment, execution times increases linearly as expected, except HyPer when going from 100M to 200M rows due to switching join algorithm. At 100M rows, all but PostgreSQL performs an in-memory join. PostgreSQL has an execution time of ±584*s* for 100M and times out at 500M rows. From 100M to 200M rows, DuckDB's execution time increases from ±15*s* to ±51*s*, HyPer's from ±29*s* to ±388*s*, and Umbra's from ±28*s* to aborting. The *Inner Width* experiment tells a similar story with expected linear scaling, although PostgreSQL manages to finish within the timeout. When going from 0 to 1 set of columns, DuckDB's execution time increases from ±12*s* to ±52*s*, HyPer's from ±27*s* to ±370*s*, and Umbra's from ±28*s* to aborting. Umbra is unable to finish any other query.

In the *Outer Cardinality* experiment, PostgreSQL has a poor algorithm choice for 200M rows but scales linearly afterward. From 600M rows onward, at least one of the five runs takes more than 1,000s for PostgreSQL. DuckDB and HyPer scale linearly as expected, although HyPer is orders of magnitude slower due to using a sort-merge join. In the *Outer Width* experiment, all systems have linear scaling as expected. PostgreSQL and HyPer barely finish with the timeout with 4 sets of columns, taking ±838*s* and ±902*s*, respectively, while DuckDB takes ±193*s*.

In the *Inner Skew* experiment, DuckDB's execution time is consistent at ±51*s*, although this decreases to ±28*s* for $\alpha = 1$ for which the distribution is Zipfian. PostgreSQL's execution time is mostly consistent, between ±518*s* and ±653*s*. HyPer's execution time drops from ±397*s* to ±32*s* when going from $\alpha = 0.5$ to $\alpha = 0.75$ because a hash table suddenly fits in main memory due to fewer unique keys. The *Outer Skew* experiment shows similar behavior, although HyPer's execution time is consistent. When going from $\alpha = 0.75$ to $\alpha = 1$, DuckDB's execution time decreases from ±51*s* to ±16*s*, PostgreSQL's from ±616*s* to ±466*s*, and HyPer's from ±377*s* to ±339*s*. Many sorting algorithms exploit skew; therefore, PostgreSQL's and HyPer's sort-merge join also benefit.

Note that DuckDB would not have been able to finish joins with larger Zipfian inner relations, as the largest hash partition must fit in RAM. DuckDB should always be able to finish joins with large Zipfian outer relations, as partitions from this site do not need to fit fully in RAM because they are read in a streaming fashion.

Finally, in the *Unique Key Count* experiment, DuckDB's execution time increases from ±51*s* for 200M unique keys to ±83*s* for 40M unique keys, as expected due to the increased number of emitted tuples. The same applies to PostgreSQL, which times out at 40M unique keys. Similar to the Inner Skew experiment, when going from 120M to 80M unique keys, HyPer switches from its external to its in-memory join due to fewer unique keys, causing execution time to decrease from ±401*s* to ±35*s*.

9 EVALUATION: PIPELINED EXTERNAL JOINS

In this section, we experimentally evaluate multi-operator memory control with workloads in which multiple joins are processed simultaneously in a pipeline. We exclude the other systems, as the previous section showed that DuckDB is the only system under benchmark that can complete larger-than-memory joins in a (partially) streaming fashion. When systems opt for an external sorting approach, only one operator is evaluated simultaneously, and multi-operator memory control becomes irrelevant.

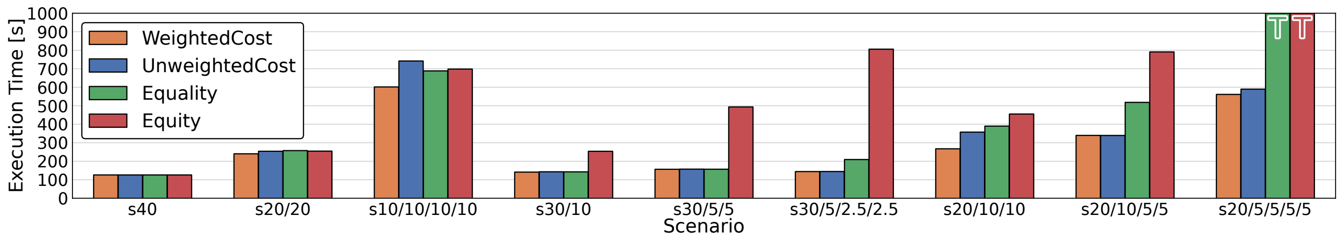

Figure 4: Execution times for join pipeline scenarios with different memory assignment policies. Lower is better. 'T' denotes that the query timed out after 1,000 seconds.

Policies. We compare four different memory assignment *policies* in DuckDB: ① *WeightedCost*: the cost model proposed in Section 6. ② *UnweightedCost*: equivalent to *WeightedCost* with weight w_i always equal to to 1. ③ *Equality*: statically assigns each operator $1/N^{\text{th}}$ of the total available memory, where N is the total number of operators. ④ *Equity*: dynamically assigns operator j with observed size s_j an amount of memory equal to s_j divided by s_{ALL} of the total available memory, where S_{ALL} is the sum of all operator sizes, i.e., Equity assigns joins memory proportional to their size.

We expect Weighted- and UnweightedCost to perform similarly. However, the width of the outer relation increases after every join by gathering columns from the inner relation. This should give WeightedCost an edge, as it considers this when optimizing the cost model. Equality and Equity should perform well when processing joins with similar sizes but poorly when they have different sizes, especially Equity, as assigning small joins less memory should increase materialization cost, as explained in Section 6.

Workload. We create nine interesting join pipeline scenarios that allow us to evaluate how the different policies behave. We only consider left-deep query plans, as only these joins are active simultaneously in DuckDB, as explained in Section 6. In these query plans, there is one outer relation and one or more inner relations. We use the same parameter configuration for the outer relation here as in the previous section: a cardinality of 500M rows, a width of 1 set of columns, skewed with $\alpha = 0.5$, and 200M unique keys. For our inner relation(s), we use a similar parameter configuration as before: a width of 1 set of columns, no skew (i.e., $\alpha = 0$). The inner relation(s) have a varying number of rows, and the number of unique keys is always equal to the number of rows.

Our first (baseline) scenario is denoted $s40$, which is a join between the outer relation and one inner relation with 400M rows. The inner relation's size is roughly equal to 2× the available memory in our experimental setup. The second scenario is a join with two inner relations, each with a cardinality of 200M rows, which we denote with $s20/20$, i.e., *we denote scenarios with the cardinalities of the inner relations* that are probed in the same pipeline. Relations are probed in the encoded order, i.e., sX/Y probes X first, then Y.

To facilitate a comparison between different scenarios, we require as an *invariant* that the sum of the cardinalities of the inner relation(s) is the same in each of the scenarios, which results in a similar space requirement for each scenario. We also use LEFT joins to prevent the outer relation's cardinality from getting reduced by joins, as this would affect all subsequent joins in the pipeline and make it difficult to compare scenarios with each other. We create seven more scenarios, for a total of nine scenarios, representing a wide variety of left-deep plans that could occur in challenging join queries. These will be explained alongside the results.

Results. We show the results in Figure 4. The first three scenarios, $s40$, $s20/20$, and $s10/10/10/10$, have equi-sized inner relations. There is no small join that should be prioritized in RAM. Instead, the best strategy is to assign a similar amount of memory to each of the joins, which all four policies do. WeightedCost has a slight edge over the other three policies, as the width of the outer relation increases after every join due to gathering the columns from the inner relation, which it takes into account by assigning slightly more memory to joins that appear later in the pipeline.

For the next three scenarios, $s30/10$, $s30/5/5$, and $s30/5/2.5/2.5$, there is a large 300M join that should be assigned less memory to allow the much smaller inner relations of ≤100M to fit in RAM. This is exactly what Weighted- and UnweightedCost do. As a result, their execution times here are only slightly higher than for $s40$. Equality also does this for $s30/10$ and $s30/5/5$, but not for $s30/5/2.5/2.5$, as it causes the 50M join to spill, degrading performance unnecessarily. As expected, Equity performs worse with each added join due to its memory assignments causing each join in the pipeline to spill.

The last three scenarios, $s20/10/10$, $s20/10/5/5$, and $s20/5/5/5/5$, have one fairly large 200M join that should be assigned less memory than the smaller joins of 100M or 50M. However, the size of the smaller joins totals 200M, meaning that they cannot all fit in RAM; therefore, assigning memory is much less straightforward than for the previous three scenarios. Equity assigns very little memory to smaller joins; therefore, it again performs worse with each added join, eventually timing out. For $s20/10/10$, the best strategy is to perform as much of the two 100M joins in memory. Equality assigns more memory to the 200M join than Weighted- and UnweightedCost and finishes in ±390s. UnweightedCost does this slightly better and finishes in ±357s, while WeightedCost finishes in ±267s. For $s20/10/5/5$, the best strategy is to perform both 50M joins in memory and as much of the 100M join as possible, leaving some space for the 200M join, which is exactly what Weighted- and UnweightedCost do. Equality takes ±180s longer, as it assigns less memory to the 100M join, causing more data to be spilled. Finally, for $s20/5/5/5/5$, Equality assigns slightly less memory to each 50M join than Weighted- and UnweightedCost; therefore, each of the four joins spills more, causing a timeout.

As expected, Weighted- and UnweightedCost perform better than the more naive policies, with WeightedCost being slightly better on some queries. It should be noted, however, that although

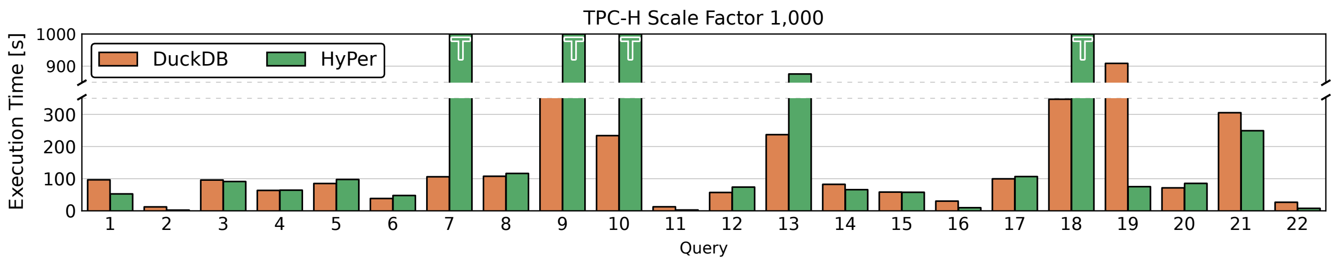

Figure 5: Execution times for TPC-H queries 1 through 22 at scale factor 1,000 (split y-axis). Lower is better. 'T' denotes that the query timed out after 1,000 seconds.

Equality and Equity are worse, they are much better than having *no policy at all*, as this would result in running out of memory and aborting the query or having to resort to processing one operator at a time, e.g., with sort-merge joins.

10 EVALUATION: TPC-H

In this section, we experimentally evaluate overall external query processing capabilities using the analytical TPC-H benchmark at SF 1,000, again using the hardware described in Section 7.

We exclude PostgreSQL from this benchmark, as it suffers from large amounts of per-tuple overhead [7] and does not unnest the correlated subqueries in TPC-H [36], causing quadratic performance. We also exclude Umbra from this benchmark, as the evaluation in Section 8 clearly shows that it cannot process larger-than-memory intermediates. Therefore, only DuckDB and HyPer are evaluated. Both implement full unnesting of arbitrary queries and have similar query plans for most queries.

Figure 5 shows the results of the benchmark. DuckDB and Hyper perform similarly for all but five queries: 7, 9, 10, 13, and 18. HyPer switches to out-of-core processing for these queries and is orders of magnitude slower than DuckDB. If we exclude these queries, the longest-running query for both systems is query 21, which takes ±183s for DuckDB and ±250s for HyPer. If we include these queries, DuckDB's longest-running query is Q9, which takes ±241s, while HyPer takes ±875s for Q13, and times out after 1,000 seconds for queries 7, 9, 10, and 18. Only queries 9, 13, and 18 require out-of-core processing for DuckDB, and despite spilling to storage, their execution time is not much higher.

Join-Heavy Queries. Queries 7, 9, and 10 have memory-intensive joins that dominate the execution time. DuckDB filters out data early in Q7 because it implements "Join-Dependent Expression Filter Pushdown" [4], which HyPer does not. However, DuckDB and HyPer have almost identical query plans for Q9 and Q10. In both plans, multiple joins are probed within a single pipeline, requiring the systems to control the memory of multiple operators. Despite having similar plans, the system's execution times are drastically different. This difference is attributed to HyPer switching to disk-based algorithms, which potentially causes it to sort the data multiple times. Meanwhile, DuckDB continues to use its hash join.

Aggregation-Heavy Queries. Queries 13 and 18 have a large, memory-intensive grouped aggregation. The plans for these queries are the same for both systems. The large aggregation is active simultaneously with a join in both queries, complicating memory control. HyPer has strong in-memory aggregation [27] but is forced to switch to a disk-based algorithm due to running out of memory. DuckDB has a similar aggregation strategy but can spill intermediates to disk [23]; therefore, it can complete much larger aggregations with many unique groups without sorting.

11 CONCLUSION & FUTURE WORK

In this paper, we have presented three techniques for external join query processing in analytical database systems, that have been implemented and released in DuckDB. We have experimentally evaluated our implementation and compared it with other systems. Our external join experiments showed that DuckDB could perform external joins much more efficiently and robustly than the other systems. Even when intermediates fit in memory, DuckDB can compete with state-of-the-art systems, showing that our external query processing techniques do not sacrifice in-memory performance. Our pipelined external join experiments showed that DuckDB's dynamic approach to multi-operator memory control assigns memory more efficiently than static or naive approaches.

Our experimental results with TPC-H at scale factor 1,000 showed that our techniques generalize to analytical workloads. DuckDB had similar execution times for queries regardless of whether it spilled intermediates to storage, showing that larger-than-memory query processing does not have to be slow. This allows larger datasets to be processed on more economical hardware, using less energy than, for example, distributed data management systems.

Future Work. Although DuckDB's hash join performs well on skewed data, *it cannot complete joins* if any of the inner relation's partitions are larger than the memory limit. If many tuples have the same join key, *repartitioning will not help.* After the initial probe, all remaining data has been materialized. Here, DuckDB could decide to *swap the inner and outer relation* depending on the sizes of the materialized partitions. Adaptively swapping sides would reduce the join's reliance on the optimizer even further and allow it to complete skewed joins that it would otherwise not be able to.

Reducing the size of intermediates, especially strings, because they are ubiquitous in real-world workloads would improve scaling further. Retaining lightweight string compression such as FSST [5] during execution could help reduce the size of hash tables even further. Retaining FSST would require holding onto some of the storage metadata during query execution and decompressing the strings as late as possible, e.g., when used in a comparison expression.

REFERENCES

[1] Josep Aguilar-Saborit, Mohammad Jalali, Dave Sharpe, and Victor Muntés Mulero. 2008. Exploiting Pipeline Interruptions for Efficient Memory Allocation. In *Proceedings of CIKM 17* (Napa Valley, California, USA) *(CIKM '08)*. ACM, New York, NY, USA, 639–648. https://doi.org/10.1145/1458082.1458169

[2] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of SIGMOD 2021* (Virtual Event, China) *(SIGMOD '21)*. ACM, New York, NY, USA, 168–180. https://doi.org/10.1145/3448016.3452831

[3] M. W. Blasgen and K. P. Eswaran. 1977. Storage and Access in Relational Data Bases. *IBM Syst. J.* 16, 4 (Dec 1977), 363–377. https://doi.org/10.1147/sj.164.0363

[4] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proceedings of TPCTC 5*, Vol. 8391. Springer-Verlag, Berlin, Heidelberg, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[5] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 12 (July 2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[6] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of VLDB 25 (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 54–65. https://doi.org/10.5555/645925.671364

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of CIDR 2005*. www.cidrdb.org, Asilomar, CA, USA, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[8] Benoît Dageville and Mohamed Zait. 2002. SQL Memory Management in Oracle9i. In *Proceedings of VLDB 28* (Hong Kong, China) *(VLDB '02)*. VLDB Endowment, Los Angeles, California, United States, 962–973. https://doi.org/10.14778/3007263.3007280

[9] Diane L. Davison and Goetz Graefe. 1994. Memory-Contention Responsive Hash Joins. In *Proceedings of VLDB 20 (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 379–390.

[10] Cedar DB. 2024. *About*. Cedar DB. Retrieved 2024-06-28 from https://cedardb.com/about/

[11] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of SIGMOD 1984* (Boston, Massachusetts) *(SIGMOD '84)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/602259.602261

[12] Thanh Do, Goetz Graefe, and Jeffrey Naughton. 2023. Efficient Sorting, Duplicate Removal, Grouping, and Aggregation. *ACM Trans. Database Syst.* 47, 4, Article 16 (Jan 2023), 35 pages. https://doi.org/10.1145/3568027

[13] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of BSDCan* (Ottawa, Canada). University of Ottawa, Ottawa, Canada, 14.

[14] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D hash join: Building on non-unique join attributes. In *Proceedings of CIDR 19*. CIDR, Chaminade, 1–9. https://madoc.bib.uni-mannheim.de/62365/

[15] H. Garcia-Molina and K. Salem. 1992. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowl. and Data Eng.* 4, 6 (Dec 1992), 509–516. https://doi.org/10.1109/69.180602

[16] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of SIGMOD 1990* (Atlantic City, New Jersey, USA) *(SIGMOD '90)*. ACM, New York, NY, USA, 102–111. https://doi.org/10.1145/93597.98720

[17] G. Graefe. 1994. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb 1994), 120–135. https://doi.org/10.1109/69.273032

[18] Goetz Graefe. 2012. New Algorithms for Join and Grouping Operations. *Comput. Sci.* 27, 1 (Feb 2012), 3–27. https://doi.org/10.1007/s00450-011-0186-9

[19] Tim Gubner, Viktor Leis, and Peter Boncz. 2021. Optimistically Compressed Hash Tables & Strings in the USSR. *SIGMOD Rec.* 50, 1 (jun 2021), 60–67. https://doi.org/10.1145/3471485.3471500

[20] Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. 2022. Design Trade-Offs for a Robust Dynamic Hybrid Hash Join. *Proc. VLDB Endow.* 15, 10 (Jun 2022), 2257–2269. https://doi.org/10.14778/3547305.3547327

[21] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of hash to data base machine and its architecture. *New Generation Computing* 1, 1 (01 Mar 1983), 63–74. https://doi.org/10.1007/BF03037022

[22] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA.

[23] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *Proceedings of ICDE 40*. IEEE, New York, NY, USA, 3753–3766. https://doi.org/10.1109/ICDE60146.2024.00288

[24] Laurens Kuiper and Hannes Mühleisen. 2023. These Rows Are Made for Sorting and That's Just What We'll Do. In *Proceedings of ICDE 39* (Anaheim, California). IEEE, New York, NY, USA, 2050–2062. https://doi.org/10.1109/ICDE55515.2023.00159

[25] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2015. Massively Parallel NUMA-Aware Hash Joins. In *In Memory Data Management and Analysis*. Springer International Publishing, Cham, 3–14.

[26] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2023. Cooperative Memory Management for Table and Temporary Data. In *Proceedings of SiMoD 2023* (Bellevue, WA, USA) *(SiMoD '23)*. ACM, New York, NY, USA, Article 2, 5 pages. https://doi.org/10.1145/3596225.3596230

[27] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of SIGMOD 2014* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[28] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[29] David Lomet. 2018. Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed. In *Proceedings of DaMoN 14* (Houston, Texas) *(DaMoN '18)*. ACM, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/3211922.3211927

[30] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of VLDB 24 (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487.

[31] Antoine N. Mourad, Robert J.T. Morris, Arun Swami, and Honesty C. Young. 1994. Limits of parallelism in hash join algorithms. *Performance Evaluation* 20, 1 (1994), 301–316. https://doi.org/10.1016/0166-5316(94)90019-1 Performance '93.

[32] Biswadeep Nag and David J. DeWitt. 1998. Memory allocation strategies for complex decision support queries. In *Proceedings of CIKM 7* (Bethesda, Maryland, USA) *(CIKM '98)*. ACM, New York, NY, USA, 116–123. https://doi.org/10.1145/288627.288647

[33] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. 1988. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of VLDB 14 (VLDB '88)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 468–478.

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

[35] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of CIDR 10*. www.cidrdb.org, Amsterdam, Netherlands. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[36] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (LNI)*, Vol. P-241. GI, Hamburg, Germany, 383–402. https://dl.gi.de/handle/20.500.12116/2418

[37] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proc. VLDB Endow.* 10, 10 (June 2017), 1022–1033. https://doi.org/10.14778/3115404.3115408

[38] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of SIGMOD 2019* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[39] Amazon Web Services. 2021. *Amazon Elastic Block Store*. Amazon. Retrieved 2024-01-22 from https://aws.amazon.com/ebs/

[40] Leonard D. Shapiro. 1986. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst.* 11, 3 (Aug 1986), 239–264. https://doi.org/10.1145/6314.6315

[41] Michael Stonebraker and Lawrence A. Rowe. 1986. The design of POSTGRES. In *Proceedings of SIGMOD 1986* (Washington, D.C., USA) *(SIGMOD '86)*. ACM, New York, NY, USA, 340–355. https://doi.org/10.1145/16894.16888

[42] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive self-tuning memory in DB2. In *Proceedings of VLDB 32* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, Los Angeles, California, United States, 1081–1092.

[43] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of ICDE 2013 (ICDE '13)*. IEEE Computer Society, USA, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[44] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of DBTest 2018* (Houston, TX, USA) *(DBTest'18)*. ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3209950.3209952

[45] Marcin Zukowski, Niels Nes, and Peter Boncz. 2008. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. In *Proceedings of DaMoN 4* (Vancouver, Canada) *(DaMoN '08)*. ACM, New York, NY, USA, 47–54. https://doi.org/10.1145/1457150.1457160