Data Chunk Compaction in Vectorized Execution

Yiming Qiao
Tsinghua University
qiaoym21@mails.tsinghua.edu.cn

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

Abstract

Modern analytical database management systems often adopt vectorized query execution engines that process columnar data in batches (i.e., data chunks) to minimize the interpretation overhead and improve CPU parallelism. However, certain database operators, especially hash joins, can drastically reduce the number of valid entries in a data chunk, resulting in numerous small chunks in an execution pipeline. These small chunks cannot fully enjoy the benefits of vectorized query execution, causing significant performance degradation. The key research question is when and how to compact these small data chunks during query execution. In this paper, we first model the chunk compaction problem and analyze the trade-offs between different compaction strategies. We then propose a learning-based algorithm that can adjust the compaction threshold dynamically at run time. To answer the "how" question, we propose a compaction method for the hash join operator, called logical compaction, that minimizes data movements when compacting data chunks. We implemented the proposed techniques in the state-of-the-art DuckDB and observed up to 63% speedup when evaluated using the Join Order Benchmark, TPC-H, and TPC-DS.

1 Introduction

Vectorized execution refers to the query processing model where each database operator computes on a vector of tuples (i.e., a data chunk) rather than a single tuple at a time. Many modern analytical databases [12, 32, 45, 46, 48] adopt vectorized execution to accelerate query processing because it reduces the interpretation overhead and improves CPU parallelism [6]. The vector size is critical to the overall query performance. If the vector is too large to fit in the CPU cache, performance will suffer from cache misses. On the other hand, if the vector contains too few tuples, vectorized execution will degenerate into the classic volcano model [13] and lose the aforementioned advantages.

Boncz et al. [6] showed empirically that the optimal size of a data chunk is in a few thousand tuples (e.g., 2048). Although we can initialize the input data chunks to this optimal size, the number of valid tuples within each chunk can be reduced by certain operators during query execution [22]. We call these operators *Chunk-Reducing Operators* (CROs). The most common CROs are filters and hash joins. After a data chunk goes through a filter operator, it updates its selection vector or bitmap and thus reduces its size effectively [30]. A vectorized hash join implementation takes a data chunk to probe hash table buckets in parallel [33]. Because each bucket can contain many items due to repeated values and hash collisions [44], an input data chunk often generates multiple (smaller) output chunks with unmatched tuples invalidated after the hash join.

We define the *Chunk-Reducing Factor* (CRF) as the chunk size entering the operator divided by the chunk size exiting the operator. We executed the Join Order Benchmark (JOB) [27] on DuckDB [36] and collected the CRF for every data chunk that goes through a hash join operator. Figure 1 shows the statistics for CRF. We observe that a majority of the data chunks become significantly smaller after a hash join, and these smaller chunks lead to increased interpretation overhead and decreased CPU parallelism for the downstream operators.

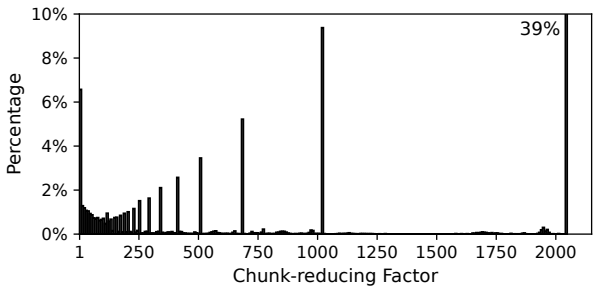


Figure 1: Distribution of the Chunk-Reducing Factor (CRF) - We executed the JOB on DuckDB and collected the CRF for every data chunk that goes through a hash join operator. The default chunk size is 2048. It shows that 39% of all the data chunks have a CRF of 2048, which means each of them contains only one record.

Therefore, compacting small chunks during execution is essential for the vectorized execution model to achieve superior performance. However, compacting data chunks involves memory copies, and such costs may outweigh the benefits of having proper-sized vectors. DuckDB handles this trade-off by predefining a size threshold $\alpha = 128$. When an output chunk contains $\leq \alpha$ valid tuples, it is copied to a buffer chunk, and the buffer chunk is sent to the next operator when it accumulates enough tuples close to its capacity (i.e., 2048 tuples). This fixed-threshold approach can be inefficient because the trade-off between interpretation overhead and memory movement is different for each chunk-reducing operator and is dependent on the number of subsequent operators in the execution pipeline. For example, there is no need to perform chunk compaction if the operator is at the end of a pipeline (i.e., a sink operator).

In this paper, we investigate *when* and *how* to compact small data chunks efficiently during query execution. We first define the chunk compaction problem and model the operator's interpretation cost and the chunk's compaction cost. We then answer the "when" question by introducing a lightweight learning-based algorithm (based on the Multi-Armed Bandit problem) to determine the compaction threshold dynamically for each CRO at run time. To approach the "how" question, we propose *logical compaction* that avoids unnecessary data movement when compacting small chunks for vectorized hash join probes. The key idea is to have separate selection vectors for the columns from both sides of the join operator within a chunk. We implemented the proposed techniques including learning-based dynamic compaction and logical compaction in DuckDB, a state-of-the-art analytical database, and

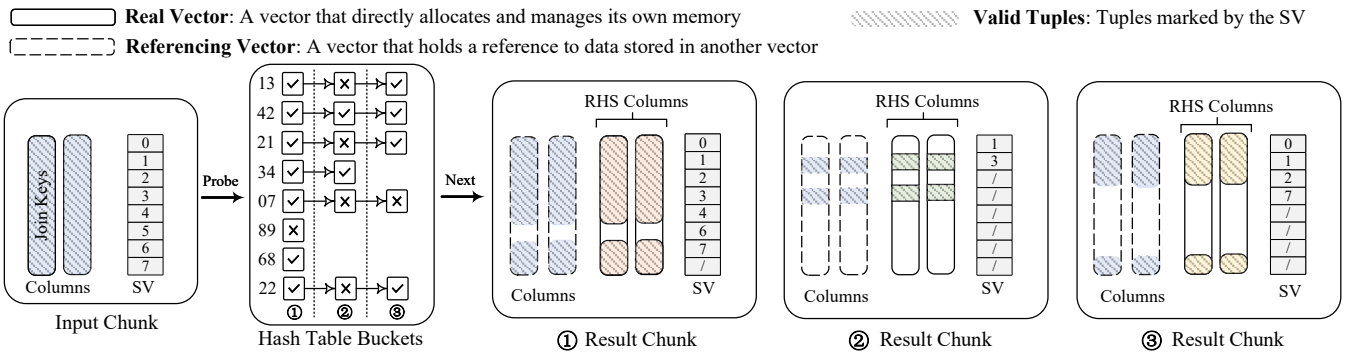


Figure 2: Vectorized Hash Join - Hashes join keys, finds matched tuples, and gathers payloads. LHS columns are zero-copy, while RHS columns require copying. We call `Next()` three times to generate three chunks, because the buckets have chains up to length three.

achieved an end-to-end speedup of 11.8%, 6.1%, and 4.6% for *all* the queries in Join Order Benchmark (JOB) [27], TPC-DS [10], and TPC-H [11], respectively. For queries where hash joins have high CRFs, the performance improvement can be up to 63% compared to the DuckDB default.

We make four primary contributions in this paper. First, we define and provide a performance analysis of the chunk compaction problem. Second, we propose an online learning-based algorithm for adjusting the compaction threshold of each chunk-reducing operator. Third, we introduce logical compaction that can minimize data movement for chunks output by hash join probes. Finally, we verify in DuckDB that the proposed solutions improve end-to-end query performance, especially for queries with multiple joins.

2 Background and Related Work

In this section, we offer the essential background on the vectorized query execution [6, 35] and the vectorized hash join operator [33].

2.1 Vectorized Model

Vectorized execution, typically implemented with the morsel-driven parallelism [26], has been widely adopted in modern analytical databases [4, 9, 12, 28, 36, 40]. The classic Volcano model [13] executes queries by calling the `Next()` interface implemented by each relational algebra operator to pull one result tuple at a time [35]. However, the fixed overhead of repeatedly invoking the `Next()` function is noticeable [18, 29]. The computational primitives in the operators must support a wide range of data types through programming techniques such as late-binding methods, function pointers, or extensive case switches, thereby introducing interpretation overhead [22, 30].

To amortize the interpretation overhead, the vectorized execution model processes a batch of tuples (e.g., 2048 tuples) for each call of `Next()`. The computational primitives are put in a tight `for` loop for the tuple batch to fully leverage the parallelism in modern super-scalar and out-of-order CPUs[14, 16]. Specifically, each execution pipeline processes a vector (i.e., data chunk) at a time without the need to materialize the intermediate results. The vector size greatly impacts query performance: too small vectors cause the engine to degenerate to the classic Volcano model, while too large vectors cause excessive cache misses. Prior studies showed that the optimal vector size is in a few thousand tuples [6].

Each data chunk uses a selection vector or bitmap to identify the valid tuples [30, 34]. For example, DuckDB [36] and Vectorwise [48] use a selection vector, while DB2 with BLU [38] employs a selection bitmap. As shown in Figure 2, the selection vector $(1, 3)$ indicates that the first and the third tuples are still valid in the data chunk. Applying a selection vector can avoid unnecessary data copy between input and output chunks, but the number of valid tuples within a chunk (i.e., the chunk size) can keep decreasing during execution.

2.2 Vectorized Hash Join

A scalar lookup in a chaining hash table involves three steps: (1) hash the tuple's join key k_1 to find the bucket; (2) compare k_1 to the first key k_2 in the bucket; (3) if $k_1 = k_2$, copy the payload to the result tuple and continue to compare k_1 to the next key in that bucket. During this processing, if the hash table does not fit in CPU cache, accessing k_2 from the bucket will cause a cache miss [42]. Such random memory accesses can easily become the performance bottleneck of a scalar hash table [22, 44]. One solution is to radix partition the table according to the join keys so that the hash table for each partition fits in cache [2, 3, 23, 39]. Although this approach reduces cache misses for hash table probes, it introduces the additional partitioning step that often dominates the join performance [2].

A vectorized hash join [26] addresses the above problems by issuing a batch of hash table probes at once to better utilize the memory bandwidth [7, 33]. Figure 2 shows an example. A chaining hash table is constructed for the right-hand-side (RHS) table. To perform a vectorized hash join, we first hash all the join keys in the input chunk from the left-hand-side (LHS) table and obtain a vector of bucket numbers $(13, 42, 21, 34, 07, 68, 22)$. We then issue a batch of memory reads to load the first item in each of the selected buckets and compare it to the corresponding input join key(s). The output is a bitmap $(1, 1, 1, 1, 1, 0, 1, 1)$ indicating if each input join key finds a match. Finally, we construct the result chunk ① by referencing the input chunk for the LHS columns (zero-copy) and gathering the payloads from the matching tuples for the RHS columns [21]. The selection vector in this result chunk is SV = $(0, 1, 2, 3, 4, 6, 7)$ because the 5th input tuple did not find a match.

We repeat the above process for the second item in each selected bucket. Again, the memory probes are issued in parallel, and the

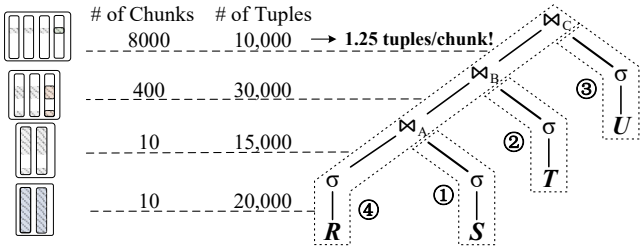


Figure 3: Pipelines of a Joining Query - The chunk becomes smaller and smaller in the probe pipeline as shown on the left.

key comparison result is $(0, 1, 0, 1, 0, 0, 0, 0)$. Consequently, the next result chunk ② contains an SV = $(1, 3)$. This batch-probing process continues until we reach the end of the bucket with the longest chain. A key characteristic of the vectorized hash join, therefore, is that an input/probing chunk can generate multiple smaller output chunks. This is unavoidable mainly because of data skew (i.e., repeated join keys) and is independent of hashing schemes.

Trade-off: Chunk Size vs. Zero-copy Benefit. To avoid small chunks, many databases, such as Apache DataFusion [24] and CockroachDB [46], sacrifice the zero-copy benefit. They copy both the LHS and RHS columns to produce full output data chunks [19, 20]. On the other hand, DuckDB and Velox prefer the zero-copy approach. The trade-off of this approach is that it can generate underfull chunks. DuckDB then predefines a size threshold and only compacts chunks with a number of tuples smaller than the threshold. In the next section, we analyze the inefficiency of current approaches.

3 The Chunk Compaction Problem

In this section, we formalize the compaction problem by analyzing the trade-off between the interpretation overhead and the data-copying cost.

3.1 Motivation

A Chunk-Reducing Operator (CRO) refers to an operator that can reduce the number of valid entries in a data chunk. The most common CROs are filters and hash joins. A query pipeline consisting of CROs can reduce the chunk sizes progressively while generating more chunks. For example, Figure 3 shows a query plan of joining four tables R, S, T, U on columns A, B, and C. This plan comprises four pipelines: three building pipelines ①-③ and one probing pipeline ④. As shown in the example, pipeline ④ receives 10 input chunks, each of 2000 tuples. The filter operator makes the data chunks 3/4 full on average. At the first hash join, the operator produces 400 chunks out of the 10 input chunks with each containing 75 tuples on average. The next join uses these 400 chunks to probe the hash table and generates 8000 chunks with a total of only 10,000 tuples, averaging 1.25 tuples per chunk. Such small data chunks, therefore, cause significant interpretation overhead for the final hash join.

Compacting smaller chunks into larger ones can reduce the interpretation overhead for subsequent operators but it involves allocating new data chunks and copying tuples from the smaller chunks into them. The decision of *when* to perform the compaction depends on balancing the overhead of interpretation and tuple copying. In general, the smaller the chunk, the greater the benefit from such compaction.

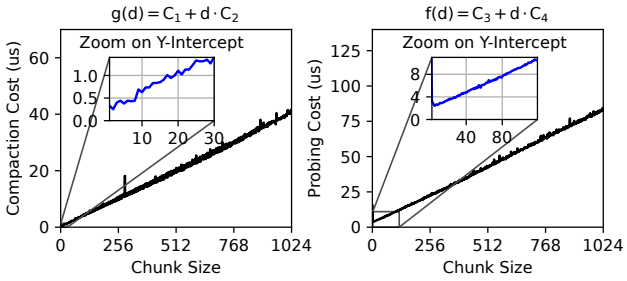


Figure 4: Parameter Profiling - We measure the fixed and per-tuple cost of chunk compaction and hash-table probing in DuckDB, getting $C_1 = 0.25\mu s$, $C_2 = 0.04\mu s$ and $C_3 = 2.4\mu s$, $C_4 = 0.08\mu s$.

3.2 Problem Formulation

Consider n data chunks with sizes $S = \{d_1, \cdots, d_n\}$, where d_i is an integer. A chunk can contain a maximum of D ($D = 2048$ by default) tuples, i.e., $1 \le d_i \le D$ for all i. These chunks are processed by a pipeline with k chunk-reducing operators. The pipeline needs time $\mathcal{F}_k(d)$ to process a chunk of size d. Let M denote a compaction on chunk set S

$$M : S \to R \triangleq \{d'_1, \cdots, d'_m\}$$

that produces chunk set R with a reduced number of chunks ($1 \le m \le n$) while preserving the total tuple count. Let $\mathcal{G}(M, S)$ denote the time required for applying the compaction M on S. The goal is to minimize the total execution time $\sum_{d' \in R} \mathcal{F}_k(d') + \mathcal{G}(M, S)$. Each operator in a pipeline encounters such a compaction challenge and must decide when to perform a compaction locally and collectively establish a globally optimal compaction policy.

Compaction Cost. Let $g(d)$ denote the time cost of a particular compaction in M where chunks $d_i, \cdots, d_j$ are compacted into a chunk of size $d \le D$. We model $g(d)$ in two parts. First, g scales linearly with the total number of tuples in the compacted chunks because of the per-tuple memory copy cost C_2. Additionally, the compaction incurs a fixed cost C_1 independent of the chunk size. Therefore,

$$g(d) = C_1 + d \cdot C_2$$

For example, as shown in Figure 4, we profiled the compaction operation in DuckDB and obtained $C_1 = 0.25\mu s$ and $C_2 = 0.04\mu s$.

Compute Cost. Let $f(d)$ denote the time needed to process a data chunk of size d by a CRO:

$$f(d) = C_3 + d \cdot C_4$$

where C_3 represents the interpretation overhead, and C_4 is the per-tuple computational cost. Figure 4 gives an example of $C_3 = 2.4\mu s$ and $C_4 = 0.08\mu s$ via profiling the hash-table probes in DuckDB.

Suppose that the number of tuples produced by each operator is $v > 0$ times the number of input tuples with a Chunk-Reducing Factor (CRF, defined in Section 1) of $r \ge 1$. For example, if $v = 2$ and $r = 4$ for an operator, then an input chunk containing 16 valid tuples will produce $16 \times 2 = 32$ tuples with each output chunk consisting of $16/4 = 4$ tuples. The total number of output chunks is thus $v \cdot r = 8$. Both filters and hash join probes are CROs. A filter operator typically generates one output chunk ($v \cdot r = 1$) for each input chunk, where $1/r$ represents the filter selectivity. On the other hand, a hash join probe can produce multiple result chunks ($v \cdot r \ge 1$) out of an input chunk, as described in Section 2.2.

The time for a k-CRO pipeline to process an input chunk of size d, therefore, is

$$\mathcal{F}_k(d) = C_3 \cdot \sum_{j=1}^{k} \min\{r^{j-1}, d\} \cdot v^{j-1} + C_4 \cdot d \cdot k \tag{1}$$

The last term is the per-tuple computing cost, which scales linearly with the number of tuples and CROs. The first term represents the interpretation cost. It is calculated based on the number of small chunks generated by the j-th operator. The j-th operator generates a total of $d \cdot v^{j-1}$ tuples, distributed across at most $(v \cdot r)^{j-1}$ chunks. Because the number of chunks cannot exceed the number of available tuples, the j-th operator outputs at most $\min\{r^{j-1}, d\} \cdot v^{j-1}$ small chunks.

3.3 A Near-optimal Greedy Strategy

In this section, we introduce a near-optimal greedy strategy, called **Sort Compaction**, that compacts small chunks aggressively whenever it is beneficial according to the models in Section 3.2. For an operator's output chunk set S, we first sort the chunks by size in ascending order (i.e., $\{d_1 \leq, \cdots, \leq d_n\}$) and create a buffer chunk $B = \{d_1\}$ with an initial size $d_B = d_1$. We then iterate the chunk list and try to decide for each chunk d_i whether to copy it into the buffer. Note that if the current buffer chunk B does not have enough capacity to hold d_i ($d_B + d_i > D$), we create a new buffer chunk that contains d_i and send the current B to the next operator as input. We define the benefit of compacting d_i into B as

$$\text{Gains}(d_B, d_i) \triangleq \mathcal{F}_k(d_B) + \mathcal{F}_k(d_i) - \mathcal{F}_k(d_B + d_i) - g(d_i)$$

where k denotes the number of subsequent operators in this pipeline. $\mathcal{F}_k(d_B) + \mathcal{F}_k(d_i)$ and $\mathcal{F}_k(d_B + d_i)$ represent the time cost without and with this particular compaction, respectively. $g(d_i)$ is the time cost for this compaction.

If $\text{Gains}(d_B, d_i) \geq 0$, we add d_i to B and update $d_B = d_B + d_i$. Otherwise, we finish the compaction as the remaining chunks are all larger than d_i, therefore, leading to lower gains. We simplify the Gains function by substituting Equation (1):

$$\text{Gains}(d_B, d_i) = \underbrace{(C_3 - g(d_i))}_{\text{Direct Gain}} + \underbrace{\left(C_3 \cdot \textstyle\sum_{j=1}^{k-1} A_j\right)}_{\text{Pipeline-level Gain}} \tag{2}$$

where $A_j = v^j \cdot \left\{\min(r^j, d_B) + \min(r^j, d_i) - \min(r^j, d_B + d_i)\right\}$. The Gains function consists of two terms: 1) the **Direct Gain**, representing the immediate benefit to the current operator from reducing its output chunks by one, and 2) the **Pipeline-level Gain**, reflecting the benefit for the subsequent operators. The term A_j quantifies the reduction in output chunks produced by the j-th operator in the pipeline when d_i is compacted with d_B.

Because of the pipeline-level gain, the position of an operator within the pipeline affects its compaction policy: the preceding operators should adopt an aggressive policy, while the subsequent operators should apply a conservative policy. Specifically, even if the direct gain for the current operator is negative, compacting d_i may still be beneficial. This is because it reduces the number of chunks that subsequent operators must process. For example, consider a pipeline with two joins ($k = 2$) where $S = \{100, 400\}$, $r = 110$, and $v = 1$. Despite a negative direct gain in Gains(100, 400) ($C_3 - g(400) = -13.85$), compaction reduces the interpretation overhead for the subsequent operators: without compaction, the 2nd join processes 210 chunks; with compaction, this number drops to 110. This reduction of $A_1 = 100$ chunks leads to a pipeline-level gain of $C_3 \cdot A_1 = 240$, resulting in a positive total gain.

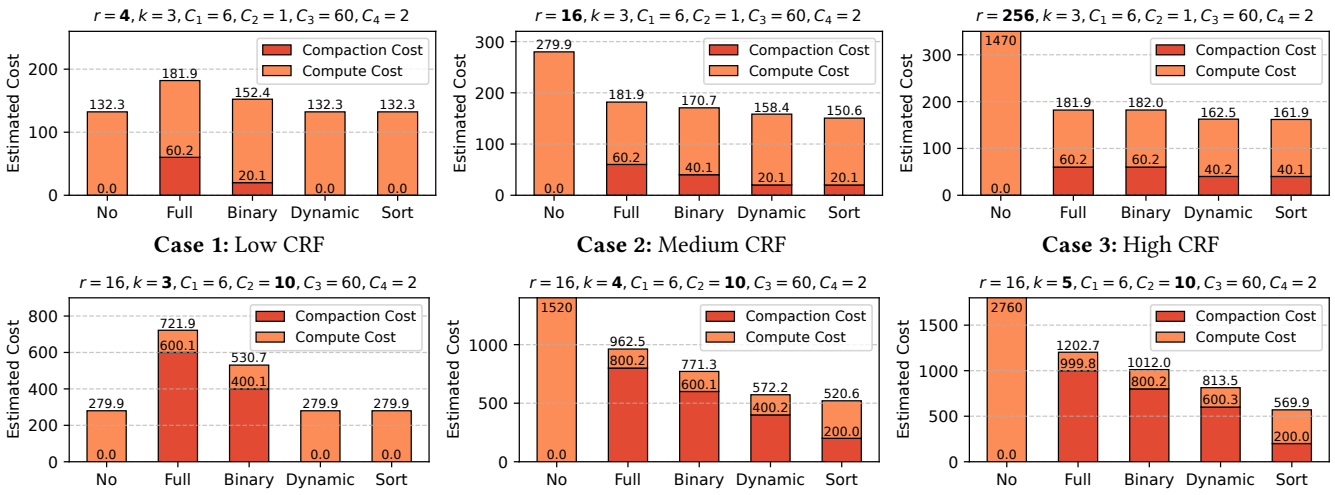
3.4 Simulation-Based Analysis

In this section, we introduce three *practical* compaction strategies. By simulating different scenarios, we assess the performance of each strategy, providing insights into their strengths and weaknesses. We consider the following strategies:

- **No Compaction & Full Compaction.** No Compaction method is the simplest approach where no chunk is compacted, while Full Compaction, on the contrary, compacts all chunks containing less than D tuples. Apache Data Fusion takes Full Compaction, which they call batch coalescence [24].
- **Binary Compaction.** DuckDB employs a compaction strategy where all chunks smaller than a predefined threshold $\hat{\alpha}$ are compacted. Given a chunk set S, DuckDB initializes an empty buffer chunk B, with size $d_B = 0$. As we iterate over each chunk $d_i \in S$, if $d_i \leq \hat{\alpha}$, we add d_i to B. When chunk B is near-full (i.e., $d_B \geq D - \hat{\alpha}$), it is sent to the next operator, and B is then reset.
- **Dynamic Compaction.** We assign each operator its own compaction threshold, computed based on Section 3.3. We assume any chunk with more than $D/2$ tuples does not require compaction. Initially, we calculate the optimal threshold $\alpha \in [0, D/2)$ such that $\text{Gains}(D/2, \alpha) \geq 0$ and $\text{Gains}(D/2, \alpha + 1) < 0$. Chunks containing $\leq \alpha$ valid tuples are then pushed into the buffer chunk, following the same procedure as the Binary Compaction.

We simulate the chunk compaction problem described in Section 3.2 by constructing a pipeline consisting of CROs, each with a fixed CRF. We set the chunk capacity to $D = 2048$ and the predefined threshold for Binary Compaction to $\hat{\alpha} = 128$. The intermediate relation produced by each operator contains the same number of tuples as the input table ($v = 1$), which contains 20 million tuples. We vary the CRF ($r = 2, 16, 256$) and the operator number ($k = 3, 4, 5$) to analyze the trade-offs between these strategies. We set parameters C_{1-4} based on the proportional relationship obtained from Figure 4. We also consider the case that a table has large tuples, leading to the increased compaction cost ($C_2 = 10$ instead of $C_2 = 1$).

In Figure 5, the first row shows that a higher CRF increases the benefit of compaction. The second row highlights the significant impact of pipeline depth on the compaction problem. Comparing Case 2 to Case 4, it is evident that for large tuples, chunk compaction hurts the overall performance because of the high costs of data copying. Figure 5 yields three conclusions. First, compaction strategies can affect query execution time significantly. Second, Binary Compaction struggles to handle diverse workloads adaptively because it relies on a predefined threshold for all CROs. Third, Dynamic Compaction, which assigns specific thresholds for each operator, demonstrates near-optimal performance. For example, in Case 1, thresholds $\vec{\alpha} = \{293, 53, 0\}$ are assigned to the pipeline's three operators, while in Case 3, $\vec{\alpha} = \{1024, 53, 0\}$ are used. It agrees with our analysis in Section 3.3: the earlier the operator is in the pipeline, the more aggressive its compaction strategy is.



Case 1: Low CRF **Case 2:** Medium CRF **Case 3:** High CRF

Case 4: Large Tuple & Shallow Pipeline **Case 5:** Large Tuple & Medium Pipeline **Case 6:** Large Tuple & Deep Pipeline

Figure 5: Compaction Simulation - Comparing compaction strategies by varying the CRF and the number of join operators.

Dynamic Compaction is hardly applicable to real databases as it relies on the function Gains to compute thresholds α, which are affected by both database design (C_{1-4}) and workload characteristics (k, r). In Section 4, we propose a learning-based approach to approximate Dynamic Compaction, estimating α without depending on these parameters. Furthermore, to minimize the substantial costs of data copying shown in Figure 5, we focus on *how* to compact chunks efficiently in Section 5 and introduce logical compaction that can compact small chunks from hash joins without data copying.

4 Learning Compaction

This section introduces a learning module designed to address the compaction problem. Although Dynamic Compaction, as previously discussed, is not feasible for real databases, this learning module serves as an approximation.

Learning-based Solution. The learning module directly estimates α from the feedback of execution (i.e., the latency). In morsel-driven parallelism, data is divided into chunks, with each thread responsible for fetching and processing a chunk through the entire pipeline before moving on to the next. Consequently, each chunk can serve as a sample for a learning algorithm [37]. Our objective is to determine the optimal threshold $\alpha \in [0, 1024)$ for each CRO. In this module, we cast the optimization of selecting the best α at runtime as a multi-armed bandit (MAB) problem [43].

4.1 Multi-Armed Bandits

The MAB problem involves a decision-maker with r options, or "arms," each with uncertain reward probabilities with an expectation μ_i. At each time step, the decision-maker selects an arm and receives a reward sampled from the associated probability distribution. The objective is to maximize cumulative reward over time. This requires the decision-maker to balance the trade-off between exploring arms to learn their rewards and exploiting arms to get high rewards.

In the pipeline context, each operator faces its own MAB problem. It selects a compaction threshold $\alpha \in [0, 2048)$ to compact its output small chunks. These chunks are then sent to subsequent operators. The reward for each arm is related to the execution latency of the pipeline and the compaction cost. Operators try different thresholds to select the optimal one, with their decisions collectively contributing to a global compaction policy.

A probing table may hold up to 20 million tuples, allowing the operator to explore arms across as many as 20,000 iterations to gather corresponding rewards. This ample sample size is adequate for fine-tuning a single parameter, α. Additionally, if operators fail to identify the optimal threshold, they can revert to Binary Compaction, using a pre-defined threshold. The performance of Binary Compaction is the lower bound of the learning approach. Lastly, the module incurs low overhead, as it merely requires statistics on the execution times of operators within a pipeline – data that modern databases already collect when profiling is enabled [36].

4.2 Online Compaction Learning

The Compaction Learner, depicted in Figure 6, optimizes the compaction trade-off during query execution. We place one compactor after each filter and hash join operator, each designated by a threshold α_i, where $1 \le i \le 3$. Before fetching a chunk from table R, the executor sets the thresholds for the compactors by invoking $\alpha_i = \texttt{SelectArm}(i)$. In response, the compaction learner provides a threshold. After processing the chunk, we measure the pipeline's latency to update the compaction learner. Specifically, the i-th compactor calls $\texttt{UpdateArm}(i, \alpha_i, t_i + \cdots + t_3)$, where α_i is the arm used in the last execution, and $t_i + \cdots + t_3$ represents the processing latency of all subsequent operators in the pipeline. This process is repeated for each chunk until all are processed.

The compaction learner maintains a statistical model for each compactor. This statistical model consists of three vectors: the candidate thresholds $\{x_1, \cdots, x_a\}$ (Arm), the estimated rewards $\{\hat{\mu}_1, \cdots, \hat{\mu}_a\}$ (Reward), and the frequency with which each threshold is selected $\{n_1, \cdots, n_a\}$ (Confidence). For instance, as illustrated in Figure 6, the model of the first compactor includes candidates: $\{0, 128, 1024\}$. Through processing chunks many times, it estimates that arm 1024 yields the highest reward. Also, the confidences of

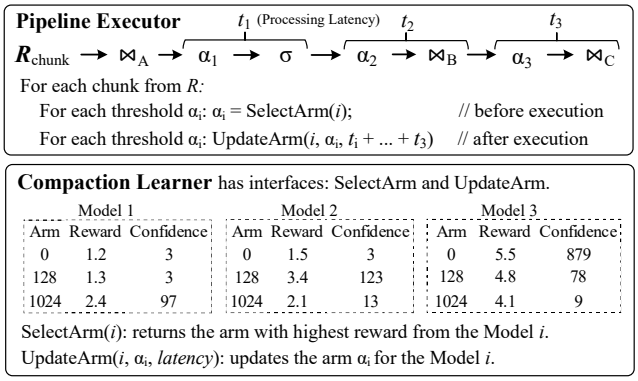


Figure 6: Learning Module Overview - Each α is a dynamic compactor, fetching the threshold from the Compaction Learner, and then updating the learner with the execution latency.

arms 0 and 128 are both 3, indicating that they have been selected only three times. This limited selection frequency may suggest a higher estimation error.

SelectArm: Exploration-exploitation Dilemma. Each model within the compaction learner confronts a dual challenge. On one hand, it seeks to select the arm with the highest reward, emphasizing exploitation. On the other hand, it must explore other arms to potentially discover more optimal solutions or to enhance the confidence in its reward estimations, particularly when the initial confidence is zero. Therefore, a well-designed policy is crucial to manage this dilemma.

The Upper Confidence Bound (UCB) algorithm is a classic approach to the multi-armed bandit problem. It selects arm x_j based on the highest value of $\hat{\mu}_j + \sqrt{(2 \ln n)/n_j}$, where $n = \sum_{j=1}^{a} n_j$ and $1 \leq j \leq a$. As the number of processing times n increases, the second term $\sqrt{(2 \ln n)/n_j}$ provides less-explored arms with a boost, even if their estimated reward is low. This property proves beneficial for balancing exploration and exploitation. A fine-tuned version of UCB takes the measured variance of rewards into account. It selects arm j based on the highest value of

$$\sqrt{(\ln n/n_j) \cdot \min(0.25, V_j(n_j))},$$

where

$$V_j(d) \triangleq \left(\frac{1}{d}\sum_{\tau=1}^{d} X_{j,\tau}^2\right) - \bar{X}_{j,d}^2 + \sqrt{(2 \ln n)/d},$$

and $X_{j,\tau}$ represents the reward of the τ-th attempt of trying arm j. This version places more trials on arms with unstable rewards.

The compactor threshold α_i lies in $[0, 1024]$. We discretize this interval and set the candidates as $\{0, 32, 64, 128, 256, 384, 512, 768, 1024\}$. This simplification reduces the parameter space without affecting overall performance because similar threshold values yield comparable compaction performance. We employ the fine-tuned version of UCB for the `SelectArm` function because of the high noise level in the pipeline execution. Additionally, at the beginning of query execution, the compaction learner attempts each arm several (e.g., 8) times to initialize their statistics. Despite resulting in a fixed overhead, this mechanism enhances the stability of the learner.

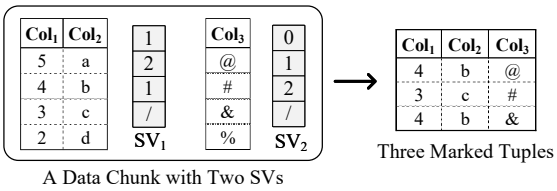


Figure 7: A Data Chunk Example with Multiple SVs - It uses SV_1 to mark the first two vectors, and SV_2 to mark the third.

UpdateArm: Taking the Moving Average. The compaction learner receives the execution latency of each chunk, measured in milliseconds, as feedback. In addressing the Multi-Armed Bandit (MAB) problem within compaction, we define the reward as the reciprocal of the latency. Initially, the rewards for all arms/thresholds are set to zero. To update the reward for each arm, we employ a moving average approach. Specifically, this involves calculating the average of the most recent 16 reward values for each arm.

One concern is that processing latency may vary greatly, while the UCB algorithm assumes that the reward should lie in a limited range. However, this is not problematic for the compaction learner focusing on the pipeline level. Typically, pipelines in most analytical queries involve less than four joins. As a result, the latencies – and consequently the rewards – received by the learner usually range from $50\mu s$ to $500\mu s$. This translates to reward values between 2 and 20, a range well-suited for effective handling by the UCB algorithm.

Additionally, to manage complex queries with potentially unstable latencies, we have implemented a robust monitoring mechanism capable of handling skewed workloads. The compaction learner periodically captures a snapshot of the estimated rewards for every η chunk ($\eta = 1024$ by default). The snapshot is denoted by $\vec{\mu} = \{\hat{\mu}_1, \cdots, \hat{\mu}_a\}$. After updating an arm x_i and obtaining a new snapshot $\vec{\mu}'$, we check for anomalies by comparing the ratios $\hat{\mu}_i'/\hat{\mu}_i$ and $\hat{\mu}_i/\hat{\mu}_i'$. If either ratio equals or exceeds 2, we identify an anomaly and reset the UCB algorithm. This reset involves clearing existing rewards and confidence levels, and each arm is tested multiple times (e.g., 8) to reestablish baseline statistics.

4.3 Multi-threading in Learning

The compaction module is designed to be compatible with morsel-driven parallelism [26]. First, a thread must acquire a lock each time it accesses the statistics from a compactor model. Second, we increment the confidence level by one, rather than in the `UpdateArm` function. This differentiation is crucial to ensure that when multiple threads (e.g., 96) simultaneously invoke the `SelectArm` function, they do not all select the same arm.

In the multi-threading environment, an interesting phenomenon occurs. Even when the `SelectArm` function chooses an arm, say j, based on the highest confidence n_j – rather than rewards – the compaction learner is still capable of providing the near-optimal arm. This happens because threads that select the optimal arm can process chunks more rapidly, leading to more frequent updates and, consequently, increased confidence in that arm. This dynamic, where each thread's update frequency effectively feeds back into the learning algorithm, is unique to morsel-driven parallelism.

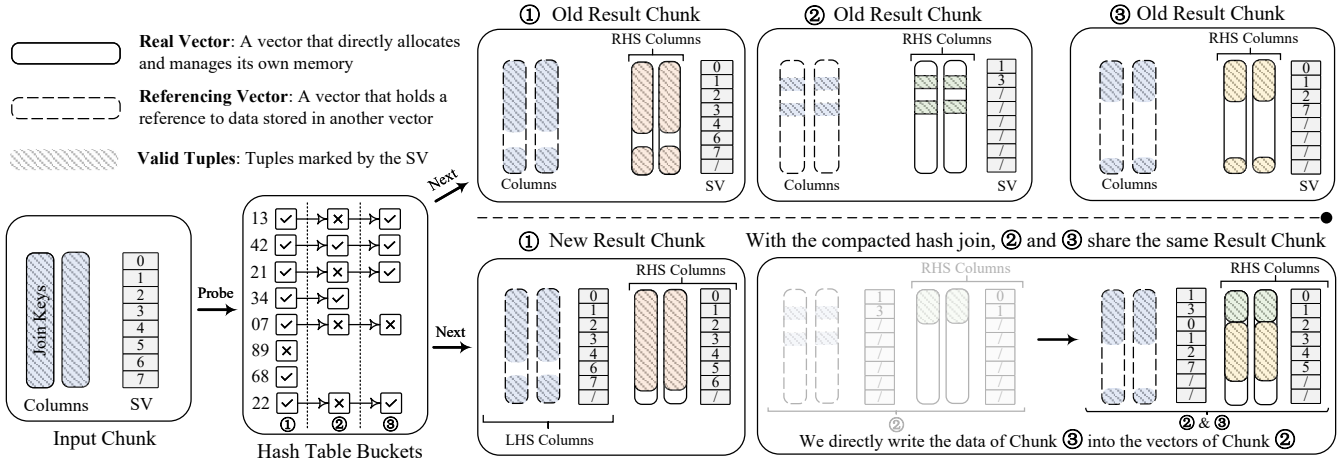


Figure 8: Compacted Vectorized Hash Join - Within each chunk, we use multiple extended SVs to manage columns from various sources. This approach enables the hash join to compact chunks without any additional cost, thus producing fewer yet larger chunks.

5 Logical Compaction

In this section, we introduce an operator called compacted vectorized compacted hash join. Figure 5 shows that the compaction cost constitutes a large portion of the overall estimated cost. To address this, we redesign the data chunk and the hash join operator. The core idea is that small chunks generated from the same probing chunk can be compacted *logically* without actual data copying.

5.1 Data Chunk Design

The current design of a data chunk comprises several data vectors and a selection vector (or bitmap), where a selection vector (SV) is a dense, sorted list of row identifiers (RIDs) indicating which tuples in the batch are valid. The DBMS marks tuples as invalid by modifying the SV alone, without copying data [30]. Figure 8 provides an example of this chunk design.

A hash join operator must modify the SV and gather the payload, to generate the result chunks. One idea is to compact chunks during the chunk generation process. Since all result chunks reference the same LHS data, we can concatenate their SVs directly. For example, we concatenate the SVs of the 2nd and 3rd old result chunks in Figure 8, setting SV = $(1, 3, 0, 1, 2, 7)$. However, this concatenation leads to RID conflicts in the SV: the term RID = 1 appears twice. The current chunk design requires that the 1st and 4th tuples are both placed in the row indexed by RID = 1. For the RHS vectors, however, placing two different values in the same place is impossible.

We propose a new data chunk design consisting of several data vectors and multiple *extended* SVs. Unlike traditional SVs, the extended SV differs in two ways: (1) it is a dense, unsorted list, and (2) it can have repeated IDs. Each SV manages a group of vectors in this data chunk. Figure 7 gives an example of a data chunk with two SVs. It uses SV_1 to mark the first two vectors and uses SV_2 to mark the third vector. Then, for the i-th valid tuple, its values in the first two columns are placed in the row indexed by $SV_1[i]$, and its value in the third column is placed in the row indexed by $SV_2[i]$. Thus, the chunk represents three valid tuples.

5.2 Compacted Vectorized Hash Join

Figure 8 illustrates the compacted hash join using extended SVs. Upon receiving a full data chunk from the LHS table, we probe its join keys, resulting in 8 hash table buckets. Subsequently, we invoke `Next()` to retrieve the output chunks.

The first output chunk contains 7 valid tuples. It uses two extended SVs: one for the LHS vectors/columns and the other for the RHS vectors. For the LHS vectors, we employ the zero-copy technique, referencing the data in the probing chunk. For the RHS vectors, payloads must be copied sequentially from the hash table, and these do not necessarily align with those in the LHS vectors of the same row. Since a chunk can hold up to 8 tuples, the 2nd result chunk, containing 2 tuples, cannot be compacted with the 1st chunk. Therefore, we directly output the first chunk as-is.

The 2nd and 3rd chunks have 2 and 4 tuples, respectively. We can compact them into a larger chunk. The second chunk follows the same format as the first, as shown by the transparent part in Figure 8. It has 2 valid tuples, with separate SVs for the LHS and RHS vectors. As for the 3rd chunk, instead of allocating new memory for it, we directly write its data into the 2nd data chunk. This decision is motivated by the fact that they hold the reference to the same vector. We can directly concatenate their SVs for the LHS columns without data copying. In Figure 8, the tuples indexed by $(0, 1, 2, 3, 7)$ are selected at least once in the SV of the 1st or 2nd chunks.

For the RHS columns, we gather the 3rd chunk's payload from the hash table and append it to the RHS vectors of the 2nd chunk, starting at index 2. We add a fully dense SV for the RHS columns in the result chunk; the shared result chunk has an SV = (0, 1, 2, 3, 4, 5) for the RHS columns because there are six valid records in total in this chunk: two entries from the 2nd chunk and four entries from the 3rd chunk. Consequently, these two chunks share the same physical memory, as shown in the rightmost, solid part of Figure 8. If more result chunks are available, we continue writing their data into the memory of the current chunk, compacting as much as possible until it can no longer hold the next chunk. Once capacity is reached, we output the current chunk and allocate a new one. Finally, the proposed compacted hash join yields two

result chunks with sizes 7 and 6, resulting in a lower CRF than the original hash join. This approach utilizes zero-copy techniques for the LHS vectors and a single gathering operation for the RHS vectors, enhancing efficiency.

Additionally, the result chunks, with multiple SVs, can be processed by the remaining operators in the pipeline. For example, consider a filter operator applied to a chunk with two SVs, $\text{sel}_1 = (1, 3, 1, 2)$ and $\text{sel}_2 = (0, 1, 2, 2)$, encompassing four tuples. Suppose the tuples at indices 0, 2, and 3 satisfy the filter criteria and are retained. This yields a result vector $\text{res} = (0, 2, 3)$. To derive the SVs for the result chunk, we update SVs using $\text{sel}'_1[i] = \text{sel}_1[\text{res}[i]]$ for $i = 0, 1, 2$, producing $\text{sel}'_1 = (1, 1, 2)$ and $\text{sel}'_2 = (0, 2, 2)$.

5.3 Overhead of Extended SVs

The compacted hash join benefits from the representation flexibility provided by the extended SVs. These extended SVs bring minimal overhead for two reasons. First, the data chunk adds only one additional extended SV when it undergoes a hash join probing. At the sink operator [26] of a pipeline, the data chunk is materialized, resetting the number of SVs to one. Since a pipeline typically has few joins, the number of SVs is not likely to become a performance bottleneck. Second, the data chunk with extended SVs maintains the same interface as the original design, avoiding the need to redesign other database components. The only change lies in determining which SV to use when accessing the column values. An optimization is that we can omit the SV from the build side after the hash join (because it is always fully dense). However, we must bring back this omitted SV if there are subsequent hash joins in the pipeline because the columns associated with the omitted SV will be on the probe side for the next join.

5.4 Column Compression

The data chunk design, consisting of several data vectors and a variable number of extended SVs, seamlessly integrates with the in-memory compressed data format during execution [1, 25]. Column compression is widely used and effective for data storage as it reduces storage size and accelerates I/O when loading data into memory. When loading the compressed data into memory, modern databases generate an in-memory data format during execution. This allows for a more compressed representation and potentially enables compressed execution throughout the system.

For example, consider a table where one of its columns is encoded using a dictionary. As we load this table into memory chunk by chunk, the vector representing the compressed column adopts dictionary encoding, referred to as a dictionary vector. In this scenario, the extended SV can function as the dictionary codes for this vector, with the dictionary serving as the vector data. Figure 9 illustrates the differences between two chunks: one without dictionary encoding (left) and the other with (right). The chunk on the left contains three uncompressed columns, while the one on the right encodes one column using a dictionary. As a result, the chunk on the right consists of two flat vectors and one dictionary vector. Note that the dictionary vector cannot logically represent elements exceeding the chunk size, aiming at enhancing cache efficiency.

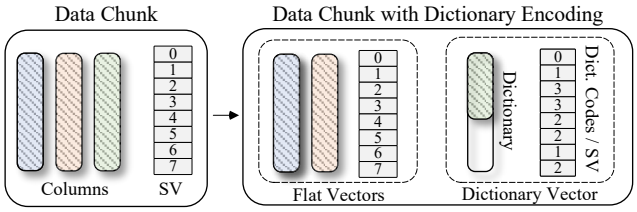


Figure 9: Compressed Vector in the Chunk - The chunk design can seamlessly integrate with column compression.

5.5 Logical and Learning Compaction

The proposed hash join method is specifically designed to compact small result chunks originating from the same probing chunk, as it requires the chunks to be compacted to share identical LHS vectors. The proposed hash join and the traditional hash join are both chunk-reducing operators, but the former has a lower CRF. The performance of the compacted hash join is significantly influenced by the chain lengths in the hash table buckets.

In a **Join-style Case**, where the operator outputs many reduced chunks from a single probing chunk, these small chunks become a performance bottleneck. The proposed hash join effectively addresses this by "compacting" the small chunks without actually performing data copying. For this reason, this method is termed *logical compaction*. Conversely, in a **Filter-style Case**, where the operator outputs only one reduced chunk from a probing chunk, the proposed hash join does not offer an advantage over the traditional method and still outputs small chunks. In such cases, we rely on the learning module described in Section 4 to tackle the issue.

Importantly, the learning module is agnostic to the specific implementation of the hash join operator; it focuses solely on compacting small chunks produced by the hash join and other chunk-reducing operators. The integration of the compacted vectorized hash join with the learning module provides a comprehensive solution to the compaction problem.

6 Microbenchmark Evaluation

We now evaluate the proposed compaction learning module, referred to as Learning Compaction, and the compacted vectorized hash join, referred to as Logical Compaction.

For the evaluation, we implement a vectorized execution engine supporting scan, filter, and hash join operators. The default size of a full chunk is set to 2048. During execution, this engine retrieves a data chunk from an in-memory data collection and processes it through a pipeline of operators. For filtering, we utilize a selection vector to mark valid tuples, and for hash joins, we implement a vectorized version similar to that in DuckDB. This hash join utilizes a chaining hash table with a load factor of 0.5. The load factor is defined as the ratio of the number of tuples to the number of buckets in the hash table. This vectorized execution engine is implemented in C++ without any explicit SIMD instructions.

Baselines. We compare our solutions to three other compaction methods: (1) No Compaction; (2) Apache Data Fusion's Full Compaction [24]; and (3) DuckDB's Binary Compaction. No Compaction is the simplest approach, where no chunk undergoes compaction. Full Compaction maintains a tuple buffer after each filter and hash join operator. It pushes any non-full chunk into this buffer and

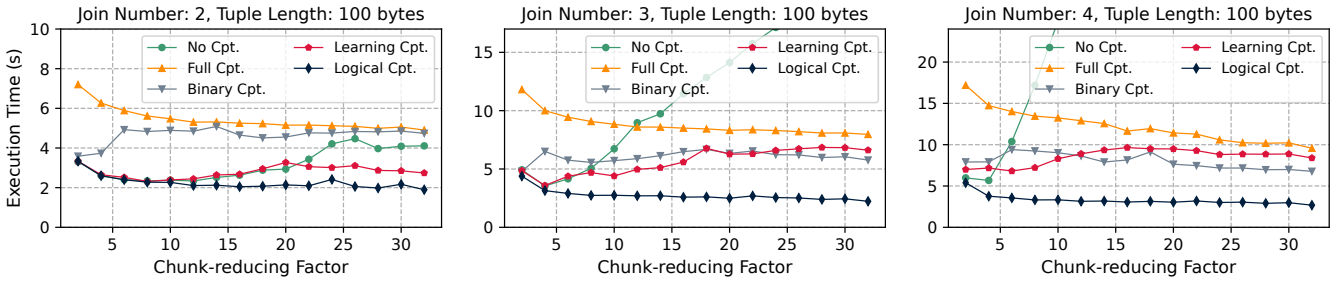


Figure 10: Execution Time vs. Chunk-Reducing Factor - This figure shows the execution times for various compaction methods across different CRFs with join numbers 2, 3, and 4. The tuple length of the probing table is fixed at 100 bytes by adjusting its string column *str*.

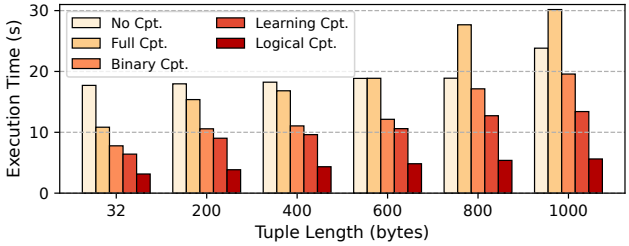


Figure 11: Execution Time vs. Tuple Length - With a join number of 4 and a CRF of 8, we vary the tuple length of the table R to demonstrate the robustness of Learning Compaction.

outputs a full chunk if there are enough tuples in the buffer. Binary Compaction also utilizes a tuple buffer but only compacts chunks containing ≤ 128 tuples, and it outputs a near-full chunk if there are ≥ 1920 tuples in the buffer.

Experimental Setup. We run experiments using our in-house server with 512 GB of DDR5 main memory at 4800 MHz and a 1 TB Intel® SSD D5-P5530. The server is equipped with two sockets of Intel® Xeon® 8474C 2.1 GHz processors (48 cores), each capable of supporting 96 threads. We use Debian GNU/Linux 12 and GCC 12.2 with `-O3` enabled. Experiments are conducted with a single thread unless stated otherwise.

6.1 Synthetic Experiment

We first evaluate these compaction methods using a left-deep join query comprising a probing side table R and k building side tables $S_1, \ldots, S_k$. The table R includes the columns $(id_1, \ldots, id_k, str)$, while each table S_i includes the columns $(id_i, misc)$. The data type of column id_i is a 64-bit integer, and the data types of *str* and *misc* are strings. By default, values in the column *misc* are fixed at 8 bytes, making each tuple in S_i 16 bytes in length. The table R has 20 million tuples, while each table S_i has 2 million tuples. The query plan performs a natural join of R with S_1 through S_k. For each join, the resulting output relation has the same cardinality as the probing relation, neutralizing the impact of intermediate result sizes. We vary the join number k from 2 to 4 and the chunk-reducing factor r from 2 to 32 in this experiment. We record the time of query execution, excluding the time spent building the hash table.

Figure 10 shows the results. First, Logical Compaction is the most efficient among these methods, providing a speedup of up to 3× compared to Binary Compaction. This can be attributed to its ability to compact most chunks without incurring data-copying costs, thereby adopting a more aggressive compaction policy. Second, when the join number is two, Learning Compaction exhibits significantly lower execution latency (by 2×) compared to Binary Compaction. This is because Learning Compaction adopts a more conservative compaction policy in this scenario. Although Binary Compaction compacts more chunks, it does not yield substantial benefits. However, for deeper pipelines or higher CRFs, the performance of Learning Compaction and Binary Compaction tends to converge. This is because deeper pipelines or higher CRFs necessitate a more aggressive compaction policy, which is precisely what Binary Compaction offers. Finally, Full Compaction proves to be more stable than the no-compaction policy. This is because the cost of compaction is generally much lower than the interpretation cost. Consequently, in scenarios involving deep pipelines and high CRFs, No Compaction suffers from higher interpretation costs.

6.2 The Compaction Trade-off

We then explore the trade-off between compaction cost and interpretation cost by varying the tuple length. We considered the following pipeline of joining five tables: `Scan(`R`) -> Join(`R`,` S_1`) -> Join(`R`,` S_2`) -> Join(`R`,` S_3`) -> Join(`R`,` S_4`)` with R having 20 million tuples and each of S_i having 2 million tuples. Each tuple in S_i has a fixed length of 16 bytes. We vary the tuple length in R from 32 bytes to 1000 bytes to show the performance impact of different compaction costs. We set the CRF to 8 in this experiment.

Figure 11 demonstrates the impact of tuple length on compaction strategies. For tuples under 600 bytes, Full Compaction is more effective than No Compaction. However, for tuples over 600 bytes, No Compaction performs better because of the high costs of copying long tuples. While the benefits of compaction remain consistent across different tuple lengths, the cost of compaction becomes higher. Additionally, Learning Compaction consistently shows lower latency than Binary Compaction for longer tuples. This is because Binary Compaction, which excels when interpretation costs dominate, applies aggressive strategies even when compaction costs are high, leading to suboptimal performance. In contrast, Learning Compaction effectively balances the compaction trade-offs, resulting in superior performance. Furthermore, as expected, tuple length has little effect on Logical Compaction.

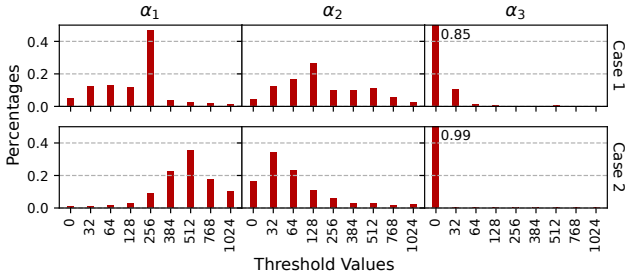


Figure 12: Distribution of Learned Thresholds - We execute a left-deep query of joining tables R, S_1, S_2, and S_3. In case 1, the tuple length of table S_2 is 16 bytes, and in case 2, it is 1000 bytes.

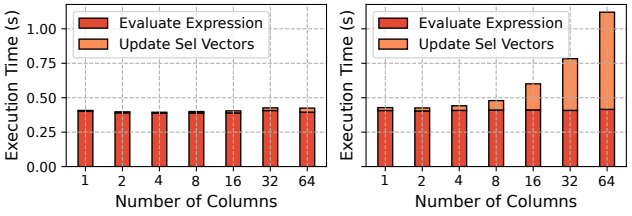


Figure 13: Overhead of Updating Additional SVs - We provide a breakdown of the filter execution. The left figure represents the single SV chunk, while the right figure represents the chunk where each column has its own SV.

6.3 Distribution of Learned Thresholds

We then show that Learning Compaction indeed learns something useful across diverse workloads. Using the same left-deep query as described in Section 6.1 with $k = 3$ and a CRF of $r = 8$, we join four tables R, S_1, S_2, and S_3. Comparing two scenarios, in the first case, tuple lengths for R, S_1, S_2, and S_3 are 32, 16, 16, and 16 bytes, respectively. In the second case, tuple lengths are 32, 16, 1000, and 16 bytes. Both cases involve three hash join operators, with the i-th operator joining its probing table with the building table S_i and employing a threshold α_i for compaction. We execute the queries with Learning Compaction and record the chosen arms for each α_i.

Figure 12 shows the learned distribution for the two scenarios. First, both cases prioritize selecting $\alpha_3 = 0$ because it is for the last join with no subsequent operators. Second, we observe the trend $\overline{\alpha}_1 \geq \overline{\alpha}_2 \geq \overline{\alpha}_3$, where $\overline{\alpha}_i$ is the most frequent chosen value for α_i. The trend aligns with the conclusions drawn from our simulations in Section 3.4: the position of an operator within a pipeline influences its compaction policy because of the pipeline-level gain. Third, in case 2, the compaction learner adapts by employing a more aggressive compaction policy for the first join, thereby alleviating compaction pressure for the subsequent join. Specifically, its α_1 is larger, and α_2 is smaller compared to Case 1. This adjustment is prompted by the presence of longer tuples in table S_2, which leads to increased compaction costs for the 2nd and 3rd joins. This comparison shows how each operator can collaborate to establish a globally optimal compaction policy. Consequently, in case 2, Learning Compaction is 1.76× faster than Binary Compaction.

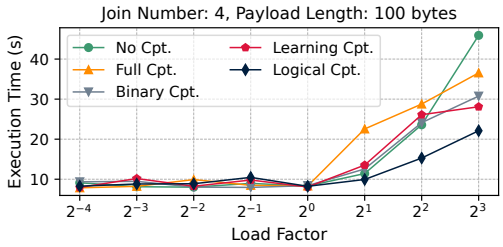


Figure 14: Load Factor vs. Execution Time - A large hash table can reduce the effect of hash collisions. With chaining hash tables, load factor = number of inserted tuples / number of buckets.

6.4 Overhead of Selection Vectors

We then show that even though the compacted vectorized hash join introduces additional SVs, they bring minimal overhead. We execute a filter on a table R with k columns $(id_1, \cdots, id_k)$, where the data type of each column is a 64-bit integer. The column id_1 contains uniformly distributed values between 0 and 100. The filter query is `SELECT * FROM R WHERE` $id_1/100 < 0.3$, with a selectivity of 0.3. We consider two chunk formats, both containing k vectors for data. The first format holds one SV for all columns, while in the second format, each column holds an SV. Thus, the filter operator needs to update all SVs for the second chunk format. Since a filter is a very lightweight operator in databases, this experiment can effectively reflect the overhead of multiple SVs. Figure 13 shows that updating SVs becomes a bottleneck when their number exceeds eight. Therefore, it is safe to have ≤ 8 SVs in a chunk. Since the number of SVs increases by one only when passing through a hash join operator, and queries with numerous joins are rare, the time cost of updating SVs will not be a performance bottleneck.

6.5 The Chain in Hash Table Buckets

A hash table bucket may have a chain of length > 1 either due to hash collisions or multiple tuples in the build table with the same key. We show that the impact of hash collisions decreases with increasing hash table size. Following the setup described in Section 6.1, we set the join number $k = 4$ and the length of each tuple in S_i to 100 bytes. We ensure that each building side table S_i has 2 million tuples. And vary the number of buckets in each hash table, such that the load factor ranges from 2^{-4} to 2^3. Figure 14 shows that we can reduce the effect of hash collisions by setting a load factor of less than 1 for the hash table. Therefore, the load factor of the hash table is set to 0.5 in our experiments.

6.6 Block Sizes

We then show the effectiveness of Logical Compaction over different data chunk sizes. We consider the same pipeline as in Section 6.1 by setting join number $k = 3$, i.e., `Scan(`R`) -> Join(`R`, `S_1`) -> Join(`R`, `S_2`) -> Join(`R`, `S_3`)` with R having 20 million tuples and each of S_i having 2 million tuples. The tuple lengths of R, S_1, S_2, S_3 are 32, 16, 16 and 16 bytes, respectively. We vary the chunk sizes to measure the execution time. We consider No Compaction, Full Compaction, and Logical Compaction in this experiment because they do not have pre-defined parameters that depend on the block size. Figure 15 shows the result, which agrees with the conclusion

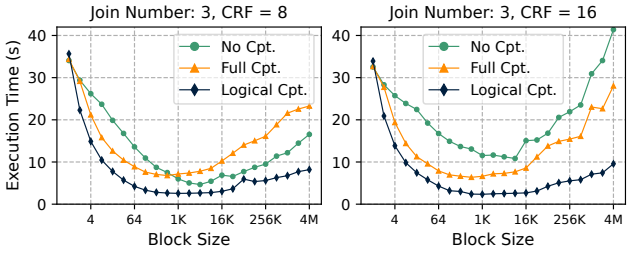


Figure 15: Varying the Data Chunk Sizes - We show the robustness of Logical Compaction by varying the data chunk sizes.

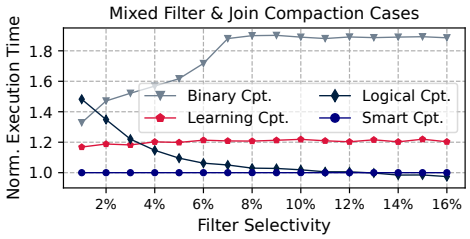


Figure 16: Filter-style vs. Join-style Case - We adjust the filter selectivity to switch from a Filter & Join style case to a pure Join-style case. Logical Cpt. is effective only for the Join-style case.

that the optimal data chunk size is in a few thousand tuples [6]. It also shows that Logical Compaction reduces the performance degradation caused by small data chunks, i.e., it flattens the curve.

6.7 Mixed Filter & Join Compaction Cases

Logical Compaction is designed for joins while Learning Compaction is mostly effective on filters. We combine them and refer to this combination approach as *Smart Compaction*. We create a pipeline containing both filters and joins and then vary the filter selectivities to show their relative significance. We first apply a filter on table R's column id_1, and then join the filtered results with table S_1, S_2 and S_3. The tuple length of R, S_1, S_2 and S_3 are 32, 2000, 16, and 16 bytes, respectively. The query pipeline is `Scan(`R`) -> Filter(`R`) -> Join(`R`, `S_1`) -> Join(`R`, `S_2`) -> Join(`R`, `S_3`)`. For each join, its CRF is set to 5. Table R contains 200 million tuples, and each S_i has 2 million tuples.

Figure 16 shows the execution time of each method, which are normalized by that of Smart Compaction. When the filter selectivity is low, the pipeline presents a Filter & Join style compaction case. Both Logical Compaction and the Learning Compaction can only handle this case partially. But their combination, Smart Compaction, can tackle the mixed-style case effectively. When the filter selectivity is high, the pipeline presents a pure Join-style case. Thus, Smart Compaction has the same performance as Logical Compaction.

7 Full DBMS Evaluation

In this section, we integrated our solutions into DuckDB (v0.8.1) [36] and measured the end-to-end performance using three benchmarks: the Join Order Benchmark (JOB) [27], TPC-H [11] and TPC-DS [10]. DuckDB is a state-of-the-art, in-process OLAP database system. It contains a columnar-vectorized query execution engine that is specifically designed to handle OLAP workloads. We integrated the compaction learning module (Learning Cpt.) into the system and replaced the default hash join operator with our compacted version (Logical Cpt.). We refer to this integrated approach as Smart Cpt.

The JOB benchmark is based on real data, specifically the IMDB dataset [41], comprising 113 multi-join queries that offer a challenging, varied, and authentic workload. In contrast, TPC-H is based on synthetic data, where the benchmark requires that data for database columns be generated from a uniform distribution. Although some columns in TPC-DS are generated using skewed distributions, the dataset still does not utilize real data.

7.1 Performance Overview

We present a performance overview of all compaction methods across three benchmarks in DuckDB, with the scale factors for TPC-H and TPC-DS set to 10. Each benchmark is run on a single thread using DuckDB's internal benchmark tools. Given the extensive number of queries in each benchmark, we measure and sum the total execution times. Figure 18 illustrates that our proposed Smart Compaction consistently and significantly outperforms both the No Compaction and DuckDB's default method, Binary Compaction, across all benchmarks. Specifically, in the JOB benchmark, Smart Compaction boosts DuckDB's performance by up to 10%.

Table 1: Profile of Hash Join Operators - We collect the runtime statistics for all three benchmarks executed by DuckDB default.

	Avg. Chunk Size	Avg. # of Chunks	Smart Cpt. Speedup
TPC-H	125.58	3.16	1.13×
TPC-DS	219.58	397.4	1.21×
JOB	54.05	689.6	1.32×

Additionally, No Compaction performs the worst. Full Compaction and Binary Compaction methods have similar performance. This indicates that the compaction trade-off is predominantly influenced by interpretation costs, similar to Case 3 in Figure 5. We note that standard benchmarks may not accurately represent real-world workloads [5, 15]. For example, none of them include tables with large tuples, which leads to high compaction costs. Despite these considerations, the Smart Compaction performs well. We then explore the underlying reasons in the following sections.

7.2 Benchmark Analysis

We perform an in-depth analysis across three benchmarks to identify the queries that benefit most from compaction strategies. By profiling each hash join operator, we evaluate two key metrics for every benchmark: (1) the average chunk size at runtime, derived from the size of all result chunks generated by hash joins, indicating the degree of chunk reduction; and (2) the average result chunk number, computed as the ratio of result chunk number to input chunk number for each hash join operator, illustrating the number of child chunks generated per probing chunk. As shown in Table 1, the speedup provided by Smart Compaction is closely correlated with these two metrics. Smaller chunk sizes and larger chunk numbers correspond to greater advantages offered by Smart Compaction. For instance, in the JOB benchmark, the average chunk size is only 54, and long bucket chains result in the generation of $\geq$ 600 child chunks from a single input chunk.

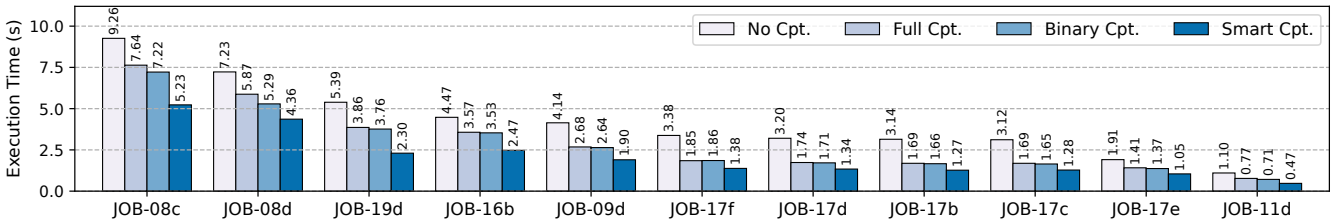


Figure 17: Execution Time for JOB Queries with Hash Join Bottlenecks - This figure illustrates the impact of different compaction methods on the performance of selected JOB queries, all of which feature hash join bottlenecks that require compaction.

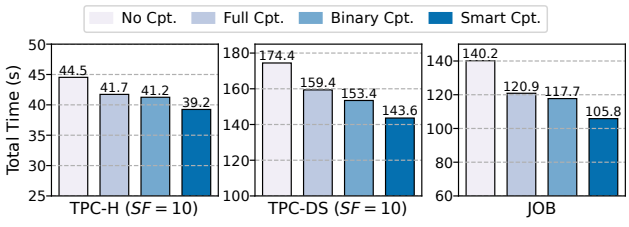


Figure 18: Benchmarks Overview - We compare the total execution times for all compaction methods on three benchmarks.

Therefore, we conclude that a query benefits from chunk compaction if it includes at least one hash join operator that: (1) constitutes a major bottleneck in the query execution, accounting for at least 10% of the total execution time; and (2) on average, generates many ($\geq$ 50) result chunks from a single input chunk. After identifying all such hash join operators across benchmarks, we select queries containing at least one qualifying operator. This process identifies 12 queries: 11 from JOB, 1 from TPC-DS, and none from TPC-H, reflecting the performance improvements seen in each benchmark. Among these, Smart Compaction offers a geometric mean performance enhancement of 34% over the standard DuckDB that employs Binary Compaction.

Figure 17 displays the selected queries from the JOB benchmark. Notably, query 19d achieves the highest speedup in our experiments, reaching up to 2.34× – a 63% improvement over Binary Compaction. Additionally, Full Compaction approaches the performance of Binary Compaction, because these queries have high CRFs, resulting in interpretation costs dominating the compaction trade-off. Smart Compaction provides distinct advantages over other methods because 1) it greatly reduces the compaction cost, and 2) estimates a specified compaction threshold for each CRO.

7.3 Case Study

We then conduct a detailed analysis of two queries, JOB 19d and TPC-H Q9, to understand the source of benefits, focusing on both the compute cost and the compaction cost.

For each hash join operator, Figure 19 displays the average number of child chunks, along with the average size of these child chunks. These child chunks are then compacted through physical memory copying before being passed to the next operator. We also record the total time spent on probing and chunk compaction.

7.3.1 JOB 19d. Query 19d from the JOB benchmark involves joining 10 tables, resulting in 9 hash join operators. DuckDB optimizes this query into a right-deep query plan, where most pipelines consist of only one join operator. This is reasonable because the JOB benchmark follows a well-defined relational schema, resulting in a joined result table smaller than other base tables. Typically, hash tables are built on the smaller tables, and since the build side is on the right, the entire plan adopts a right-deep style [27].

In this query, the hash join operators exhibit a high CRF and long bucket chains. Traditional hash join operators, in Binary Compaction, produce many small result chunks. The compacted hash join, utilized in Smart Compaction, effectively decreases the number of result chunks and increases their sizes, offering significant improvements over traditional hash joins.

The 6th hash join operator is the only one whose average chunk size decreases when using Smart Compaction, compared to the default Binary Compaction. This anomaly occurs because: 1) this hash join operator has a bucket chain length of one, presenting a filter-style compaction case, as introduced in Section 5.5; and 2) the input chunks to the 6th operator are smaller when Smart Compaction is employed, as the preceding operator compacts fewer chunks before the 6th hash join compared to Binary Compaction.

7.3.2 TPC-H Q9. Most queries in the TPC-H benchmark have joins with relatively short chains. Among these, Query 9 exhibits the longest chain. Query 9 comprises five hash join operators, with only one of them benefiting from Smart Compaction, achieving a speedup of 1.18×. For the remaining joins, both Smart and Binary Compaction exhibit similar performance. This result is expected because the TPC-H benchmark's average chain length is only 3.16. Notably, Smart Compaction reduces the number of results chunks to one for all hash joins, demonstrating its effectiveness. Our experiment reveals a significant gap in data distributions between synthetic benchmarks (TPC-H, TPC-DS) and real-data benchmarks (JOB), which substantially impacts the compaction problem.

Summary. This case study yields three key conclusions. First, Real-data workloads (JOB) have significantly smaller average chunk sizes than synthetic benchmarks (TPC-H). Second, small chunks are a major latency bottleneck, especially for hash join operations. Third, Smart Compaction outperforms Binary Compaction by reducing data-copying overhead, using adaptive compaction thresholds at runtime, and the compacted vectorized hash join.

7.4 Relative Significance

We then show the relative significance of proposed techniques by categorizing queries into four groups: 1) benefiting only from Logical Compaction; 2) benefiting only from Learning Compaction; 3) benefiting from both, and 4) benefiting from neither. As described in Section 6.7, Logical Compaction is designed for joins while Learning Compaction is mostly effective on filters. Figure 20 shows the

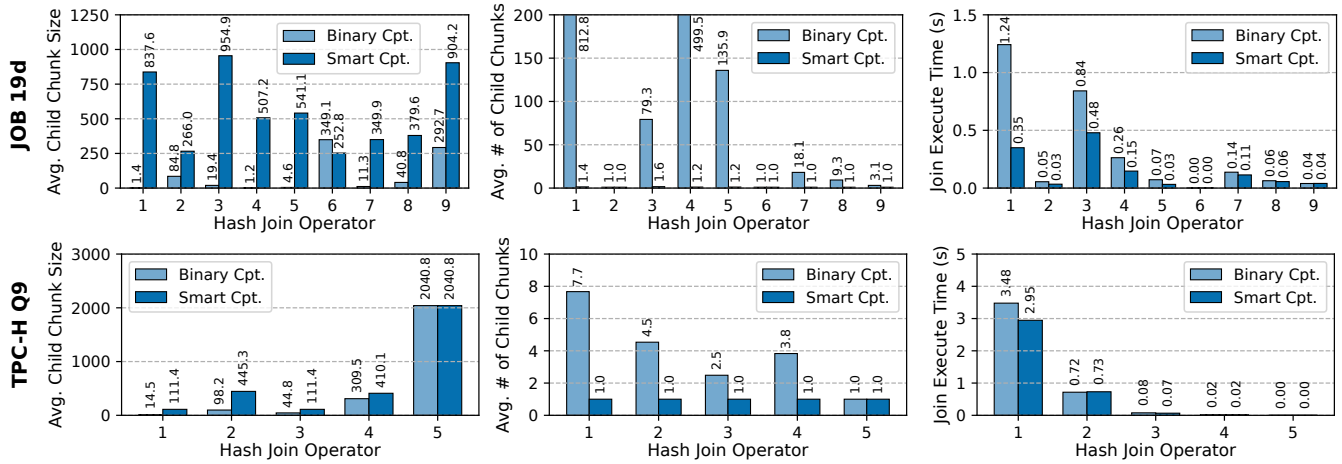


Figure 19: Case Study – A Profile of Joins - For each hash join operator, we measure the average size of output chunks, the average number of child chunks generated by a single input chunk, and the execution time. We select two representative queries from a real-data benchmark (JOB) and a synthetic benchmark (TPC-H).

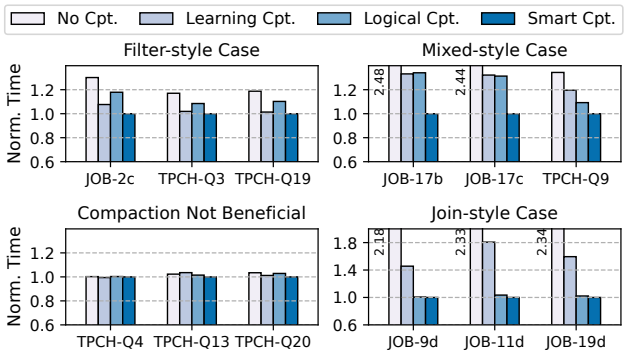


Figure 20: Relative Significance of Learning and Logical Compaction - We select queries to show various compaction cases. Execution time of each query is normalized by that of Smart Cpt.

results. For example, TPC-H Q19, a filter-style query, has two filters (with selectivities of 0.05% and 14%). Its join operators produce near-full chunks, and thus Logical Compaction is ineffective. In join-style queries, Smart Compaction performs similarly to Logical Compaction, as the selective joins, not filters, present a performance bottleneck. In mixed-style queries, Smart Compaction significantly outperforms both individual techniques, confirming the need to combine them to handle this case effectively, consistent with our analysis in Section 6.7. Finally, some queries gain no benefit from compaction, due to the absence of selective filters and joins.

8 Discussion

Most modern analytical database engines today adopt the vectorized execution model (e.g., ClickHouse [40], DataFusion [24], Photon [4], SnowFlake [12], and Velox [32]). Data chunk compaction is a general problem for these engines. For example, Velox has an open issue on optimizing the performance with small chunks [17]. DataFusion handles this problem by introducing a pre-configured switch to select between Full and No Compaction [31].

The vectorized hash join algorithms in many systems [8, 24, 32, 46] were derived from MonetDB/X100 [6, 47]. Therefore, our proposed Logical Compaction also applies to these implementation variants. Systems such as DataFusion and CockroachDB copy both the probe- and build-side columns from the input chunks to the result chunks when performing the hash join [19, 20]. This is equivalent to the Full Compaction strategy in our paper. On the other hand, DuckDB avoids copying the probe-side columns by including references to the input chunks [21]. Although this approach significantly reduces the memory copy cost, the side-effect (identified in this paper) is that it can easily generate under-full chunks. Logical Compaction solves the chunk compaction problem for vectorized hash joins so that the "zero-copy" approach consistently exhibits performance advantages over Full Compaction.

The applicability of Logical Compaction is independent of whether the table is partitioned because partitioning (e.g. radix partitioning) happens before executing the vectorized hash join. It is also independent of the hash table types (e.g., chaining vs. open-addressing) because Logical Compaction is carried out on the already gathered matched tuples. Note that if the data chunk implementation uses bitmaps instead of selection vectors (SVs), Logical Compaction must convert the bitmaps to SVs (this can be done efficiently using vectorized instructions) before the compaction.

9 Conclusion

In this paper, we formalized the chunk compaction problem, which involves balancing data copying costs and interpretation costs in vectorized query execution. We proposed learning compaction, which enables the dynamic adjustment of compaction policies during runtime. Additionally, we introduced logical compaction that can compact data chunks without actual data copying for vectorized hash joins. Our investigation reveals that learning compaction effectively addresses the *when* to compact challenge, while logical compaction improves *how* to compact. We integrated both methods into DuckDB and evaluated their performance against JOB, TPC-H, and TPC-DS. The results show that our proposed techniques achieve up to 63% performance improvement over the default DuckDB.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of SIGMOD'16*. ACM, 671–682.

[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'21*. ACM, 168–180.

[3] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. 2017. Distributed Join Algorithms on Thousands of Cores. *Proceedings of VLDB'17* 10, 5 (2017), 517–528. https://doi.org/10.14778/3055540.3055545

[4] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of SIGMOD'22*. ACM, 2326–2339.

[5] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proceedings of TPCTC'13erformance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8391)*. Springer, 61–76.

[6] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of CIDR'05*. www.cidrdb.org, 225–237.

[7] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. 2023. Analyzing Vectorized Hash Tables Across CPU Architectures. *Proceedings of VLDB'23* 16, 11 (2023), 2755–2768.

[8] Angela Chang. 2019. 40x faster hash joiner with vectorized execution. https://www.cockroachlabs.com/blog/vectorized-hash-joiner/

[9] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-Ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil Mckay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *Proceedings of VLDB'19* 12, 12 (2019), 2022–2034.

[10] The Transaction Processing Council. 2021. TPC-DS Benchmark (Version 3.2.0).

[11] The Transaction Processing Council. 2022. TPC-H Benchmark (Version 3.0.1).

[12] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of SIGMOD'16*. ACM, 215–226.

[13] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.

[14] Philipp M. Grulich, Aljoscha P. Lepping, Dwi Prasetyo Adi Nugroho, Varun Pandey, Bonaventura Del Monte, Steffen Zeuch, and Volker Markl. 2023. Towards Unifying Query Interpretation and Compilation. In *Proceedings of CIDR'23*.

[15] Andrey Gubichev and Peter A. Boncz. 2014. Parameter Curation for Benchmark Queries. In *Proceedings of TPCTC'14Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8904)*. Springer, 113–129.

[16] Tim Gubner and Peter A. Boncz. 2021. Charting the Design Space of Query Execution using VOILA. *Proceedings of VLDB'21* 14, 6 (2021), 1067–1079.

[17] Optimize Operator's Performance When Vector has Low Selectivity. 2023. https://github.com/facebookincubator/velox/issues/7801

[18] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[19] Apache DataFusion Hash Join Implementation. 2024. https://github.com/apache/datafusion/blob/f7efd2d31adb51a67dc6bfb6d6eae6a525d60482/datafusion/physical-plan/src/joins/utils.rs#L1223

[20] CockroachDB Hash Join Implementation. 2024. https://github.com/cockroachdb/cockroach/blob/67e99ebec74c1f6a6dfbf1cc0bca2d255a55f867/pkg/sql/colexec/colexecjoin/hashjoiner.go#L631C3-L631C16

[21] DuckDB Hash Join Implementation. 2024. https://github.com/duckdb/duckdb/blob/e2b177b759dbb7cabae0c0afd041bb7de2a9e698/src/execution/join_hashtable.cpp#L928

[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proceedings of VLDB'18* 11, 13 (2018), 2209–2222.

[23] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of VLDB'09* 2, 2 (2009), 1378–1389.

[24] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Proceedings of SIGMOD'24*. ACM, 5–17.

[25] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of SIGMOD'16*. ACM, 311–326.

[26] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of SIGMOD'14*. ACM, 743–754.

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of VLDB'15* 9, 3 (2015), 204–215.

[28] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proceedings of VLDBV'20* 13, 12 (2020), 3461–3472.

[29] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of VLDB'17* 11, 1 (2017), 1–13.

[30] Amadou Ngom, Prashanth Menon, Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, and Andrew Pavlo. 2021. Filter Representation in Vectorized Query Execution. In *Proceedings of DaMoN@SIGMOD'21*. ACM, 6:1–6:7.

[31] Configuration Settings of DataFusion. 2024. https://datafusion.apache.org/user-guide/configs.html

[32] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proceedings of VLDB'22* 15, 12 (2022), 3372–3384.

[33] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of SIGMOD'15*. ACM, 1493–1508.

[34] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards Practical Vectorized Analytical Query Engines. In *Proceedings of DaMoN@SIGMOD'19*. ACM, 10:1–10:7.

[35] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of SSDBM'16*. ACM, 16:1–16:12.

[36] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of SIGMOD'19*. ACM, 1981–1984.

[37] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In *Proceedings of SIGMOD'13*. ACM, 1231–1242.

[38] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proceedings of VLDB'13* 6, 11 (2013), 1080–1091.

[39] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of SIGMOD'16*. ACM, 1961–1976.

[40] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proceedings of VLDB'24* 17, 12 (2024), 3731–3744.

[41] IMDb Data Set. 2024. https://www.imdb.com

[42] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of VLDB'94*. Morgan Kaufmann, 510–521.

[43] Aleksandrs Slivkins. 2019. Introduction to Multi-Armed Bandits. *Found. Trends Mach. Learn.* 12, 1-2 (2019), 1–286.

[44] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of DaMoN@SIGMOD'11*. ACM, 33–40.

[45] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of VLDB'05*. ACM, 553–564.

[46] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of SIGMOD'20*. ACM, 1493–1509.

[47] Marcin Żukowski et al. 2009. *Balancing vectorized query execution with bandwidth-optimized storage*. SIKS.

[48] Marcin Zukowski and Peter A. Boncz. 2012. From x100 to Vectorwise Opportunities, challenges and things most researchers do not think about. In *Proceedings of SIGMOD'12*. ACM, 861–862.