Selective Late Materialization in Modern Analytical Databases

Yihao Liu
Tsinghua University
liuyihao24@mails.tsinghua.edu.cn

Shaoxuan Tang
Tsinghua University
tsx23@mails.tsinghua.edu.cn

Yulong Hui
Tsinghua University
huiyl22@mails.tsinghua.edu.cn

Hangrui Zhou
Tsinghua University
zhouhr23@mails.tsinghua.edu.cn

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

Late Materialization (LM) is a critical technique applied in traditional column stores to speed up analytical queries. However, with modern analytical databases evolved to incorporate a vectorized columnar execution engine, LM's benefits in I/O reduction and fast columnar query processing have diminished. In this paper, we redefine the concept of Late Materialization in the context of modern analytical databases and propose Selective Late Materialization (SLM) to allow each attribute in a query to choose its own materialization point that yields the minimum cost. SLM expands the solution space of the traditional materialization problem from one unified hard-coded binary decision (i.e., early or late) for all attributes to per attribute per query decisions. By integrating SLM into DuckDB, we show that SLM consistently outperforms the baselines of Early Materialization and Late Materialization by 14.7% and 8.9%, respectively, on average using the Join Order Benchmark (JOB), with up to 76.7% latency reduction for individual queries. We observe similar results for the TPC-DS benchmark.

PVLDB Reference Format:
Yihao Liu, Shaoxuan Tang, Yulong Hui, Hangrui Zhou, and Huanchen Zhang . Selective Late Materialization in Modern Analytical Databases. PVLDB, 18(11): 4616 - 4628, 2025.
doi:10.14778/3749646.3749717

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yhliu918/duckdb/tree/latest.

1 INTRODUCTION

Column Stores are predominant in analytical query processing. They organize data in columnar format so that queries only read columns needed from storage to reduce unnecessary I/Os. Late Materialization (LM) is a common technique applied in traditional column stores to improve query performance [10, 12]. It is widely adopted in mainstream database management systems, including C-store [47], MonetDB [18, 39], VectorWise [50], Vertica [31], Umbra [16, 40], Redshift [15], and DataFusion [32]. According to [12], LM avoids combining the columns into rows (i.e., tuple reconstruction) early in the query pipeline. Instead, it reads and processes columns independently and passes virtual tuple IDs as intermediate results. By delaying tuple reconstruction to the latest operator possible, LM reduces data movement from disk to memory and improves CPU and cache efficiency via columnar execution.

However, the execution paradigm in modern analytical databases (e.g., DuckDB [42], Prestissimo [41], ClickHouse [44], Redshift [15]) has evolved since Late Materialization was first proposed in traditional column stores [24, 25]. Today's analytical systems adopt a vectorized columnar execution engine, where every operator in the query plan process data in batches. Each batch is a horizontal slice of the table and is internally organized as column vectors, similar to PAX format [14]. Consequently, the systems no longer need Late Materialization to enable columnar execution, and they no longer perform explicit tuple reconstruction (unless for the final output) required by traditional column stores.

The evolution of the execution model compels us to revisit the concept of Late Materialization and reevaluate its performance trade-offs in modern analytical databases. In this paper, we first analyze why Late Materialization (LM) loses its performance edge over Early Materialization (EM) in a vectorized execution engine. First, the advantages of columnar execution, such as fast value iteration via vectorized processing, are inherent in modern execution engines, no matter what materialization strategy they adopt. Second, while LM can reduce I/O by skipping values that are already filtered out, the performance gain becomes smaller with modern NVMe SSDs as they offer much higher bandwidth than a decade ago. In many cases, such I/O reduction does not justify the cost of inefficient I/O patterns caused by the out-of-order probing problem in LM. Our microbenchmark in Section 5 verifies that applying LM to DuckDB does not improve query performance on average, compared to the default EM strategy.

Nevertheless, Late Materialization still speeds up operators that (1) are highly selective, and/or (2) carry row-oriented payloads (e.g., in a hash table). LM can skip a large portion of the storage blocks when processing highly selective operators, and it can reduce memory copies of the payload columns between row-oriented data structures and data vectors. Therefore, instead of hard-coding the materialization strategy to the execution engine, we propose **Selective Late Materialization** (SLM) that can determine the optimal materialization point for each attribute involved in a query independently based on a trained cost model.

Selective Late Materialization opens two degrees of freedom rarely explored in prior work. First, unlike traditional EM/LM, which requires every attribute in the query to adhere to the same materialization strategy, SLM allows each attribute to pick a different strategy to minimize its own induced cost of I/O and memory copy

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749717

in the execution. Second, SLM considers every operator between the traditional EM point (i.e., table scan) and LM point (i.e., where the values are needed for computation) as a feasible materialization point to accommodate cases where the optimal materialization point is in the middle (e.g., when the attribute cardinality decreases and then increases between the EM and LM points.)

Given an attribute in a query, SLM first identifies all its feasible materialization points on the query's physical plan. Then, SLM computes the materialization cost for each candidate point using our cost model, consisting of fetching and memory copy costs. The parameters in the model are trained/calibrated using sample queries. Finally, SLM modifies the physical plan and assigns the materialization point with the minimum cost to the attribute.

We integrated SLM into DuckDB and evaluated its performance against the traditional EM and LM strategies. Our results on the Join Order Benchmark (JOB) show that SLM achieves an average speedup of 14.7% (max 37.6%) over EM. Compared to LM, the average speedup of SLM is 8.9% (max 76.6%). For TPC-DS, SLM is 10.9% faster than EM on average (max 20.7%) and is 10.3% faster than LM on average (max 73.0%) Moreover, SLM exhibits robustness in the experiments by delivering a performance close to the optimal materialization strategy (i.e., obtained via an exhaustive search) in every evaluated query.

Our paper makes the following contributions. First, we identify the key reasons why Late Materialization in modern analytical databases is no longer as beneficial as in traditional column stores. Second, we redefine the concept of Materialization and propose Selective Late Materialization (SLM) that allows each attribute in a query to choose the optimal materialization point independently. Finally, we integrate SLM into DuckDB with a minimal change of its architecture and demonstrate SLM's performance gains and robustness in microbenchmarks and end-to-end evaluation.

2 REVISITING LATE MATERIALIZATION

Late Materialization (LM) has significantly improved the performance of traditional column stores by minimizing I/O and memory operations. It achieves this by allowing direct operations on (compressed) columnar data and reconstructing as few tuples as possible. However, modern analytical databases have evolved to incorporate a vectorized columnar execution engine with a PAX-like storage and memory format [1, 14, 33]. This requires reevaluating the performance trade-offs of applying LM to modern analytical systems. Section 2.1 reviews LM's definition and trade-off analysis in traditional column stores. Section 2.2 describes how the LM techniques adapt to modern execution models. Section 2.3 redefines LM under the context of modern analytical databases and provides motivations for Selective Late Materialization (SLM).

2.1 Late Materialization in Column Stores

Column stores organize data in a column-wise manner on disk. They not only avoid unnecessary I/Os for irrelevant columns but also allow lightweight compression [10, 11, 37] and vectorized execution [18] to speed up query processing. Early column stores [10, 24, 25] store data column-wise to enable selective column reading, but convert to row tuples for execution. Seminal definitions of Early Materialization (EM) and Late Materialization (LM) describe

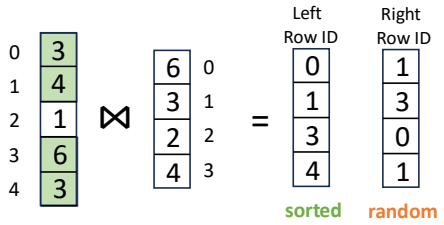


Figure 1: The out-of-order probing problem.

materialization as reconstructing tuples from columns [12]. To compare fairly between EM and LM, the above work extended C-Store with row-oriented execution with a merge operator for tuple construction and enabling row-based operators. Materialization, by then, includes two major steps: (1) fetching column values from disk or memory buffer, and (2) assembling row tuples (intermediate result or final output). Under this definition, Early Materialization reads all the columns needed by the query during the table scan and performs tuple reconstruction immediately. In contrast, Late Materialization only reads columns needed by the current operator and delays tuple reconstruction to the latest possible point in the execution. Many later works [20, 26, 28] follow the above definition.

Late Materialization performs selective operations such as filter and join before reconstructing tuples. These operations only scan relevant columns and output virtual tuple identifiers (e.g., row IDs). At tuple reconstruction, LM rescans the table and fetches the values according to the identifiers. According to the analysis in [12], LM offers two primary benefits compared to EM: (1) speeding up operators (e.g., filter and join) by taking advantage of columnar processing techniques such as vectorized execution and operating directly on compressed data, and (2) reducing I/O and memory footprint by scanning only the qualified tuples during reconstruction. However, LM could incur high overhead during tuple reconstruction (i.e., fetching values based on row IDs) because of unfriendly I/O patterns. As shown in Figure 1, for example, the right-hand-side (i.e., build-side) row IDs after a join are likely to have a random distribution, leading to the out-of-order probing problem later at tuple reconstruction.

While later column stores evolve to process columns directly for many operators, they often hardcode late materialization strategies. For example, C-Store [47] defaults to materializing after filtering, MonetDB [39] defers materialization until output (a pure LM strategy), and Vertica [46] applies LM only to probe-side attributes.

2.2 Modern Vectorized Analytical Databases

Compared to early column stores where the concept of Late Materialization originated, modern analytical databases such as Velox [41], DuckDB [42], StarRocks [8], and Apache Doris [6] adopt a vectorized execution engine where data flows between operators in small columnar batches (typically with several thousand rows). We follow DuckDB's terminology and call this format DataChunks. Each column in the DataChunk is associated with a Selection Vector (SV), a set of DataChunk-local row IDs to indicate the valid rows. Multiple columns in the DataChunk can share the same SV if they have the same valid row IDs. A database operation, such as filter and join,

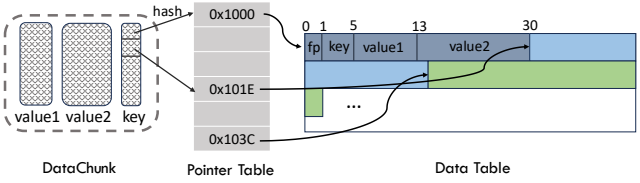


Figure 2: Row-based Hash Table Layout.

would operate directly on the relevant columns in the DataChunk and update the corresponding SV(s) to record the results.

The above execution paradigm shift has changed the definition and trade-off analysis of Early/Late Materialization. First, modern analytical systems no longer perform the "assembling row tuples" step (i.e., the column-to-row conversion) during tuple reconstruction (unless for the final output) because all the intermediate results are represented uniformly using DataChunks. As a result, the only remaining step of materialization in modern analytical databases is to replace the virtual tuple IDs with real values on disk or in the memory buffer. Therefore, the trade-off analysis between LM and EM now concentrates on whether the I/O reduction achieved by minimizing value substitutions can outweigh the cost of inefficient I/O patterns caused by the out-of-order probing problem.

Second, almost all the operators in modern analytical databases adopt and benefit from vectorized columnar processing regardless of the choice of materialization strategies. Consequently, unlike in traditional column stores, LM has limited advantages over EM in speeding up operator computations. However, there are still operators that use row-oriented data structures to store the payloads to reduce random memory accesses. The most important operator in this case is Hash Join. Figure 2 illustrates the hash table layout in a hash join used by DuckDB [42] (and other systems such as Velox [41]). The hash table is essentially a pointer array, each pointing to a list of items in the Data Table. Each item contains the join key, a 1-byte fingerprint, and other attribute values as the payload and is stored contiguously in a row so that it only requires a single memory access to retrieve the payload during the join. By replacing the attribute values with virtual tuple IDs (i.e., Late Materialization) in the hash table, the system can reduce the size of payload memory copies during both hash table construction and probing. A smaller hash table also reduces memory footprint and improves cache performance.

2.3 The Need for Selective Late Materialization

Based on the analysis in Section 2.2, we redefine the process of Materialization in modern analytical databases as *replacing the virtual tuple IDs with real attribute values in column vectors at some designated point during query processing*. The above analysis also implies that the performance advantage of applying Late Materialization decreases in modern analytical databases, compared to that in traditional column stores. The benefit of reducing the I/O size diminishes as modern NVMe SSDs offer much higher bandwidth than a decade ago.

Additionally, the explicit column-to-row conversion only happens in a few critical operations, such as hash join. Figure 3 shows a summary of our evaluation (detailed in Section 5) of Early (EM)

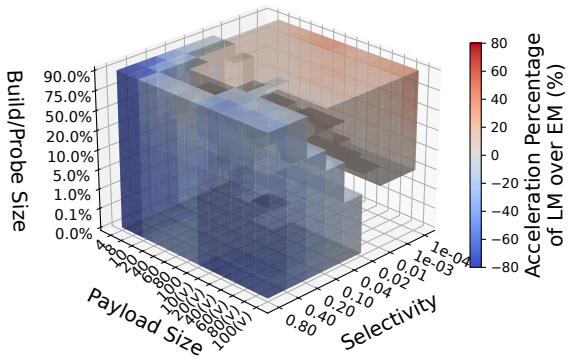


Figure 3: Performance overview of LM versus EM. – To highlight performance boundaries, regions with <5% difference are omitted. Payload size with suffix "(v)" denotes variable-length strings.

versus Late Materialization (LM) for single build-side attributes in hash joins under various configurations. We found that LM excels only with highly selective operators and those carrying a long row-formatted payload. Through a comprehensive microbenchmark analysis in Section 5, we conclude that unlike in traditional column stores where LM often dominates, neither pure EM nor pure LM is universally optimal for modern analytical databases.

Instead of eagerly applying LM to all the attributes at the latest materialization points, we propose *Selective Late Materialization* (SLM) where the materialization strategy for each attribute in the query is determined independently based on a trained cost model. The traditional EM describes the earliest materialization point during the initial table scan, while the traditional LM represents the latest materialization point when values are first used for computation. Any operator in between is a valid materialization point. The goal of SLM is to choose the materialization point for each attribute to minimize the performance cost of query processing.

We next introduce the Selective Late Materialization algorithm in Section 3. Section 4 describes how to integrate SLM to modern vectorized execution engines in general as well as implementation details in DuckDB.

3 SELECTIVE LATE MATERIALIZATION

The core idea of Selective Late Materialization (SLM) is to select the materialization point for each attribute in the query independently to minimize the execution cost relevant to this attribute. Algorithm 1 illustrates the procedure. The first step is to determine the feasible materialization points for each attribute in the query's physical plan. The *earliest* point to materialize an attribute is at its corresponding base table scan, while the *latest* point is when the attribute is first used for computation (e.g., as a filter column or a join key). Then, for each feasible materialization point for the attribute, we compute its **Materialization Cost** $Mat(\cdot)$, consisting of two components: **Fetching Cost** $Fetch(\cdot)$ and **Memory Copy Cost** $Mem(\cdot)$. Fetching Cost represents the estimated time of retrieving the attribute values using row IDs and populating them into DataChunks at the materialization point, while Memory Copy Cost refers to the estimated time of copying the payload in each

Algorithm 1 Selective Late Materialization (SLM)

Input: Physical Plan
Output: {attr → mat_point}
1: **for** each *attr* ∈ {attributes in the query} **do**
2: *earliest* ← the scan operator for *attr*'s base table
3: *latest* ← the point *attr* is first used in computation
4: *root* ← root operator of the plan
5: {mat_points} ← [*earliest*, *latest*) ▷ All feasible mat points
6: {ops} ← [*earliest*, *root*] ▷ All operators involving *attr*
7: {mcOps} ← operators in {ops} requiring memory copy
8: **for** each p_i ∈ {mat_points} **do**
9: $C_{\text{fetch}} \leftarrow Fetch(p_i)$ ▷ Fetching Cost
10: $C_{\text{mem}} \leftarrow \sum_{op \in \{mcOps\}} Mem(op)$ ▷ Mem Copy Cost
11: $C_{\text{Mat\_i}} \leftarrow C_{\text{fetch}} + C_{\text{mem}}$
12: mat_point ← p_i with minimum $C_{\text{Mat\_i}}$
13: **return** {attr → mat_point}

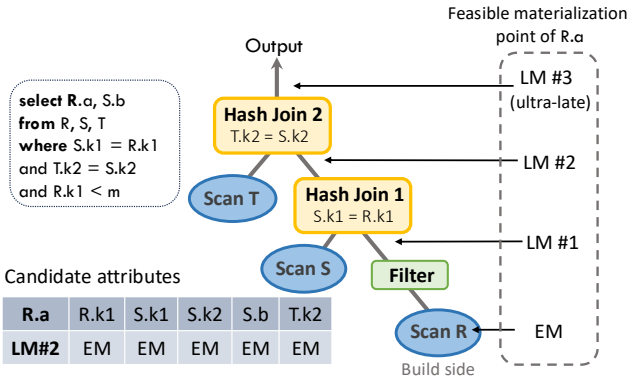


Figure 4: An example of Selective Late Materialization (SLM).

operator that computes on a row-oriented data structure (e.g., hash join), as discussed in Section 2.2. Finally, the SLM algorithm outputs the materialization point with the minimum Materialization Cost and assigns the point to the attribute.

Figure 4 shows an example query plan involving two hash joins and six attributes. k_1, k_2 are the join keys, and a, b are the payloads. Because $S.k_1$ and $T.k_2$ are needed to compute the joins right after their corresponding base table scans, they only have one feasible materialization point. We continue to call the materialization point at table scan (i.e., *earliest* in Algorithm 1) Early Materialization (EM), following the conventional definition. Any materialization point later than EM is considered Late Materialization (LM). Each of the remaining four attributes ($R.k_1$, $R.a$, $S.k_2$, and $S.b$) has multiple feasible materialization points. For example, the join payload $R.a$ has four candidate points (EM, LM#1, #2, and #3), as shown in Figure 4. The SLM algorithm, therefore, computes the Materialization Cost for each candidate point and picks the one with the lowest cost (i.e., LM#2) as the materialization point for attribute $R.a$.

In the rest of this section, we present the Materialization Cost model. Section 3.1 discusses the Fetching Cost ($Fetch(\cdot)$) of existing and our optimized fetching algorithms. Section 3.2 models the Memory Copy Cost ($Mem(\cdot)$) introduced by tuple reconstruction in operators that adopt row-oriented data structures.

3.1 Fetching Algorithms and Costs

As defined in Section 2.3, materialization in modern analytical databases refers to the process of fetching the attribute values using the row IDs. Therefore, its cost depends on the fetching algorithm. Given a list of row IDs, a fetching algorithm includes two steps: (1) performing disk I/Os to load the needed storage blocks to memory, and (2) populating the values in the storage blocks to the corresponding DataChunks for query processing. Based on the row ID patterns (i.e., sorted or in random order), we apply different fetching algorithms and model their Fetching Costs accordingly.

3.1.1 Sorted row IDs. When an attribute is first loaded into a DataChunk through a table scan, it is either Early Materializationed or represented as a sorted list of virtual tuple identifiers (i.e., row IDs). The row IDs will remain sorted before the materialization point unless the attribute encounters a shuffling operation, such as the build side of a hash join that randomly distributes the row IDs to the result DataChunks. At the materialization point, if it is the EM point or the input DataChunks contains a sorted list of row IDs of the target attribute, the system performs sequential I/Os to fetch the storage blocks (use zone maps for I/O skipping), extracts the values indicated by the row IDs, and then writes them to the result DataChunks. Because both disk and memory access patterns are sequential, we model the Fetching Cost $Fetch(\cdot)$ as $T_S(N, \#S)$, a linear combination of the number of row IDs (N), and the number of storage blocks loaded ($\#S$).

3.1.2 Random row IDs. Operators such as a hash table build can shuffle the row IDs, causing the out-of-order probing problem at the materialization point (refer to Section 2.1). In this case, an efficient fetching algorithm is critical to achieving a low Fetching Cost. We first present two baseline algorithms: **naive** and **sort**. We then describe our optimized **batch** algorithm for better cache efficiency.

naive: This algorithm issues a fetch operation (i.e., loads the storage block and extracts the target value) for each row ID in tuple access order. When the row IDs are randomly distributed, the algorithm generates excessive random disk I/Os, destroying the query performance.

sort: This algorithm [36, 48] sorts all the row IDs for the target attribute at the materialization point before issuing any fetch operations. Therefore, it changes the costly random disk I/Os above to fast sequential accesses. After fetching the values from storage, the algorithm needs to restore the original order to the values before populating them to DataChunks. Although the sort algorithm achieves an efficient I/O pattern, the computational overhead of global sorting and order restoration is too high to be practical in most situations.

batch: As illustrated in Figure 5, given the row IDs from a set of DataChunks, the Batch algorithm retrieves the attribute values in three phases. First, the *Batch* phase groups the row IDs whose values are in the same storage block. Then, the *Fetch* phase loads the storage blocks within a group via sequential scan and extracts the values associated with the row IDs to an intermediate buffer. Finally, the *Output* phase maps the values in the intermediate buffer back

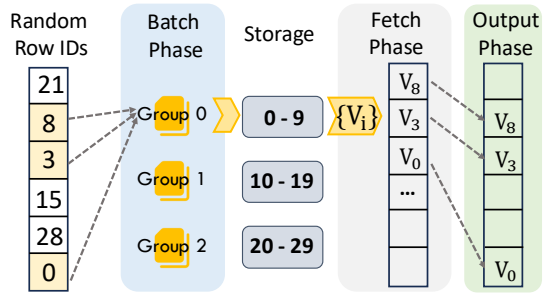


Figure 5: An overview of the Batch fetching algorithm.

to the DataChunks. However, existing Batch algorithms operate on a complete list of row IDs whose size typically exceeds CPU cache [38, 45]. Both Batch and Output phases need to reorder the row IDs (or store explicit mappings between row IDs and output locations) and, therefore, generate excessive cache misses that slow down the fetching process.

We, therefore, introduce an optimized Batch algorithm to keep the working set properly sized to minimize cache misses. The goal is to restrict the peak memory usage of the algorithm to the (last-level) cache size **C**. The algorithm first performs the Batch phase as row IDs to be materialized continue to arrive. We store the row ID $\rightarrow$ output index mapping for each group using a simple vector of pairs V[1]. The Batch phase monitors two sizes: $\mathcal{V}(t)$ and $\mathcal{F}(t)$, where t denotes the total number of row IDs accumulated so far. $\mathcal{V}(t)$ represents the size of the map V, while $\mathcal{F}(t)$ calculates the peak memory footprint in the Fetch phase obtained from the most "memory-consuming" group. Whenever $\mathcal{V}(t) \geq \eta\mathbf{C}$ or $\mathcal{F}(t) \geq \eta\mathbf{C}$ (e.g., $\eta = 0.9$), the algorithm enters the Fetch phase to retrieve the attribute values. The Batch phase freezes until the current Fetch phase is completed. In this way, we guarantee that the working set always fits in the LL cache.

Suppose group i contains t_i row IDs. Assume the base table of the target attribute contains m storage blocks, each with $\mathcal{B}$ tuples. Then the memory footprint of group i in the Fetch phase is:

$$\mathcal{F}(t_i) = \mathcal{V}(t_i) + \mathcal{B} \cdot p + t_i \cdot p \tag{1}$$

where p denotes the attribute length. In the above equation, $\mathcal{B} \cdot p$ represents the size fetched from storage, and $t_i \cdot p$ represents the size of the intermediate/output buffer. In our optimized algorithm, the Output phase is unnecessary because random memory writes occur when writing results back becomes cheaper as the results remain in the cache.

Based on the optimized Batch algorithm, the Fetching Cost is modeled below. We first model the latency of a single Batch phase $T_B(\cdot)$ and Fetch phase $T_F(\cdot)$ for processing n rows as input. $T_B(\cdot)$ is determined by n and the number of storage blocks in the attribute's base table m. For $T_F(\cdot)$, two additional features influence its latency: the physical size of the attribute p, and the materialization point's cardinality N. We employ a Gradient Boosting Regressor to model the relationship between these features and the phase latencies, expressed as $T_B(m, n)$, and $T_F(m, N, n, p)$.

Next, we estimate the number of times these phases are triggered during execution. The maximum number of row IDs $\hat{t}_i$ within a single group that would trigger the Fetch phase is derived from the memory constraint $\mathcal{F}(\hat{t}_i) < \eta\mathbf{C}$ in Equation (1). Assuming row IDs are randomly distributed across the m storage blocks, the total accumulated number of row IDs that are processed in one run of the Batch and Fetch phase is $\hat{t} = m \cdot \hat{t}_i$. Thus, the Batch and Fetch phase are repeated $\lceil \frac{N}{\hat{t}} \rceil$ times. When multiple late-materialized attributes $\{p_i\}$ originate from the same base table, they share a single Batch Phase but undergo individual Fetch Phases. In total, the Fetching Cost is modeled as:

$$Fetch(\cdot) = \lceil \frac{N}{\hat{t}} \rceil * (T_B(m, \hat{t}) + \sum_{p \in \{p_i\}} T_F(m, N, \hat{t}, p)) \tag{2}$$

In real-world datasets, row IDs may exhibit non-uniform distribution across storage blocks. This skewness leads to $\hat{t} < m \cdot \hat{t}_i$, causing our model to overestimate $\hat{t}$ while underestimating $\frac{N}{\hat{t}}$. However, our experiments in Section 5.9 demonstrate that the observed row ID skewness in public benchmarks introduces only minor deviations (<5%) in fetching cost at most materialization points. We, therefore, conclude that the uniform row ID assumption remains effective for cost estimation in most practical scenarios.

3.2 Modeling Memory Copy Cost

The Memory Copy Cost includes the tuple construction overhead in all operators that employ row-oriented data structures along path from the attribute's base table scan to the query output. Take the Hash Build operator as an example. According to the hash table layout in Figure 2, we model the Hash Build time $\mathcal{HT}(\cdot)$ based on: (1) the build-side table size$\|R_B\|$, (2) the total payload size $\mathcal{P} = \sum p_i$, and (3) the total memory copy size, including copying payloads to the *data table* and allocating the *pointer table*. We, therefore, model the Hash Build operator op^i_{ht} using the following linear model:

$$\mathcal{HT}(op^i_{ht}) = a_0 * \|R_B\| + a_1 * \mathcal{P} + a_2 * (\|R_B\| \cdot (1 + \mathcal{P}) + 2\|R_B\| \cdot 8)$$

The last term is the total memcpy size, with $2\|R_B\| \cdot 8$ representing the size of the pointer table, which is an 8-byte pointer array with a length twice that of the build-side table.

For attributes whose lengths exceed that of row IDs (typically 4-byte integers), replacing their values with row IDs in operators such as Hash Build preceding the materialization point can reduce the total memory copy size. Suppose the total payload size in hash build operator op^i_{ht} shrinks from $\mathcal{P}$ to $\hat{\mathcal{P}}$ by storing row IDs, the hash table construction can be accelerated by $\Delta\mathcal{HT}(op^i_{ht}) = (a_1 + a_2 * \|\mathcal{R}_{\mathcal{B}}\|) * (\mathcal{P} - \hat{\mathcal{P}})$.

4 INTEGRATION WITH MODERN DATABASES

This section describes how to integrate Selective Late Materialization into modern analytical databases. First, Section 4.1 provides preliminaries of the pipeline execution mechanism. Then, Sections 4.2 to 4.4 describes in detail the SLM optimization workflow, as overviewed in Figure 6. Given a physical plan, we first enumerate all attributes systematically to identify their feasible materialization points (Section 4.2). For each attribute, we compute

[1]An alternative is to use a hash table. Our evaluation in Section 5.2 shows that the de-duplication benefit of using a hash map [7] does not justify its construction/operation overhead compared to a simple array.

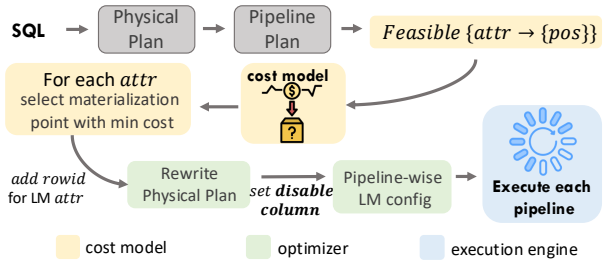


Figure 6: Workflow outline.

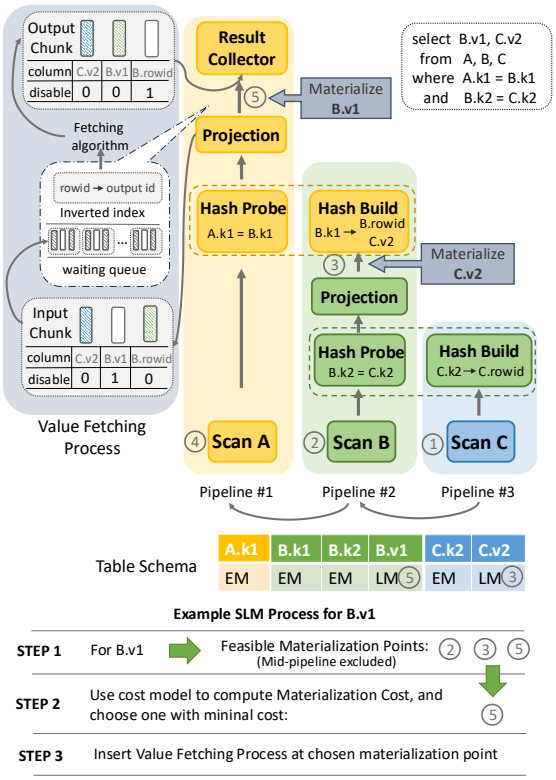


Figure 7: Supporting Selective Late Materialization.

the Materialization Cost at each feasible materialization point and pick the one with the lowest cost. Then, the optimizer modifies the physical plan by replacing attribute values with their corresponding row IDs before each attribute's materialization point (Section 4.3). Finally, Section 4.4 describes the changes made to the execution engine to support SLM in each execution pipeline. We describe the implementation details of applying SLM to DuckDB in Section 4.5.

4.1 Preliminaries of pipeline execution

Modern analytical databases (e.g., Velox [41], StarRocks [8], Doris [6], DuckDB [42]) often adopt pipelined query execution paired with morsel-driven parallelism [34]. Tables are partitioned into small DataChunks, which are combined with the corresponding query fragment (i.e., pipeline) to form tasks. Tasks are then dynamically scheduled across worker threads to enable fine-grained load balancing and better CPU utilization. Each pipeline consists of (1) a *source* operator that retrieves DataChunks from storage or upstream pipelines, (2) an optional sequence of ordinary operators that process DataChunks synchronously (without I/O or network operations), and (3) a *sink* operator acting as a pipeline breaker (e.g., hash build) that requires consuming all input Data-Chunks before producing output. If an inter-pipeline dependency exists (e.g., when a sink's output feeds another pipeline's source), the dependent pipeline must be scheduled after its predecessor completes.

4.2 Identifying Feasible Materialization Points

We restrict materialization to occur only at the *source* operator or right before the *sink* operator of each pipeline. For each attribute, we identify all potential materialization points by traversing the operators from its source table scan to its first active use (as a join key, aggregation key, grouping key, filter predicate, output column, etc.). We tailor the potential points using the above restrictions to obtain the final feasible set. For example, in Figure 7, attribute $B.v_1$ has feasible materialization points at its Table Scan (②), the *sink* of pipeline #2 (③), and the *sink* of pipeline #1 (⑤)

We exclude mid-pipeline operators because materialization invokes the blocking/asynchronous I/O operations, which violates the principle of pipelined execution. However, our implementation still supports materialization at arbitrary mid-pipeline operators by inserting a pipeline breaker to create two dependent sub-pipelines and performing materialization at the breaker. Our empirical study in Section 5.8 shows that mid-pipeline materialization is effective only when cardinality at the mid-point is significantly smaller than that at the source or sink, which rarely occurs in public benchmarks.

After finding all the feasible materialization points, the cost model extracts features from each point and selects the one with minimal Materialization Cost.

4.3 Generating Materialization Plan

We next describe how to generate physical plans with chosen materialization points. Attributes exist as row IDs before their materialization points and are converted to actual values thereafter. To achieve this, the row IDs of late-materialized attributes are added to the SQL projection list, and the physical planner generates an updated plan that maintains both the actual value columns and their associated row ID columns for these attributes. Materialization at specific points is controlled using a `disable_column` identifier within DataChunks to indicate whether a column will participate in subsequent operator computations. For example, in Figure 7, before the value fetching process of $B.v_1$, we enable its row ID column; after materialization, we enable its value column while disabling the row IDs. Note that a row ID column could be shared by multiple value columns in a DataChunk. Therefore, it is only disabled when all the value columns have been materialized.

Additionally, we modified Table Scan and Hash Build operators to handle disabled columns. Table Scan omits I/O for disabled columns while generating row IDs when present. Similarly, Hash Build excludes disabled columns during hash table construction (e.g., in Figure 7, it processes only B.rowid and $C.v_2$, skipping disabled $B.v_1$). All other operators process DataChunks normally.

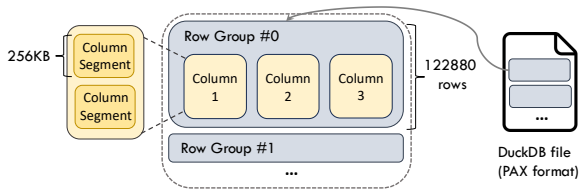


Figure 8: The hierarchical memory management of DuckDB.

4.4 Materialization in a Single Pipeline

As shown in the value fetching process in Figure 7, to support materialization within a single pipeline with minimal changes to its execution logic, DataChunks reaching the *sink* operator are first placed into a waiting queue. We perform the Batch Phase of the fetching algorithm on this queue with the corresponding row ID column. We then trigger the Fetch Phase and write the results into a pre-allocated Vector. Finally, the materialized value column in the output DataChunks extracts a slice from this result Vector before feeding it into the *sink*.

4.5 Implementation Details in DuckDB

We integrated SLM into DuckDB for an end-to-end evaluation in Section 6. In this section, we describe additional implementation details in DuckDB. We introduce four pipeline control signals to enable the above SLM procedure:

- PUSH_SOURCE: if true, pass attribute base tables to parent pipelines.
- MAT_STRAT: trigger value fetching if late materialization exists.
- SOURCE_ID (optional): map attributes to base tables.
- MAT_MAP (optional): map row IDs to value columns in the DataChunk preceding the *sink* operator.

As shown in Figure 8, DuckDB's two-tier memory structure (Row Groups: 120K rows; Column Segments: 256KB) allows deploying the **batch** fetching algorithm at either level. I/O skipping occurs at the Column Segment level, and DuckDB's `FetchRow` API facilitates row-level value fetching. Column Segment-level batching improves locality during the Fetch Phase but requires binary search in the Batch Phase due to variable record counts in different segments (e.g., with `varchar` attributes or compression). Row Group-level batching avoids binary search but may incur random accesses across different Column Segments during the Fetch Phase. For simplicity, we default to batch at the Row Group level because it typically fits within the L2 cache. However, Column Segment-level batching may be considered for large payloads exceeding L2 cache capacity.

Although DuckDB currently uses synchronous I/O in Table Scan operators, the batch fetching algorithm inherently supports async I/O, because it organizes the unpredictable disk accesses caused by random row IDs into predictable sequential accesses.

5 MICROBENCHMARK

In this section, we present a comprehensive microbenchmark for a single hash join under various parameters to analyze the trade-offs between EM and LM. This motivates the proposed Selective Late Materialization technique, which is evaluated in Section 6.

Name	Meanings	Value options
$\|\|R_P\|\|$	Probe side table size.	20M rows.
$\|\|R_B\|\|$	Build side table size.	$[0.1, 1, 10, 20, 50, 90]\% * \|\|R_P\|\|$
sel	Join selectivity	$[0.01, 0.1, 1, 2, 4, 10, 20, 40, 80]\%$
$type_p$	Payload data type.	`int32`, `int64`, fixed/variable length string of/up to [10, 20, 40, 60, 80, 100] B.
p	Payload size ($\|\|type_p\|\|$).	[4, 8, 10, 20, 40, 60, 80, 100] B.
HIT_DIST	Hit frequency distribution of probe side keys in B_{hit}.	Uniform or Zipfian ($\alpha = 1.5$)
B_PAT & P_PAT	Positions of hit keys in build/probe side table. This parameter influences the pattern of result row IDs.	[Random, Sorted, Clustered]. Hit keys are in random/sorted order in the column. Clustered: only $\%_{b\_hit}$ or $\%_{p\_hit}$ of storage blocks contain hit keys.
$\%_{b\_hit}$ & $\%_{p\_hit}$	Portion of storage blocks that contain hit keys.	$[0.01, 0.1, 1, 2, 4, 10, 20, 40, 80, 100]\%$. $\%_{b/p\_hit} = 100\%$ if B/P_PAT!=Clustered

Table 1: Different configuration in microbenchmark.

5.1 Experiment Setup

Our microbenchmark includes multiple single-join queries, each in the form of `SELECT attr_under_study FROM build, probe WHERE build_key = probe_key;` We vary the types and sizes of the payload attributes to study the trade-offs between EM and LM in DuckDB.

5.1.1 Data sets. We generate a $20M$-row probe table R_P and a smaller build table R_B, each including a 4-byte integer join key and multiple payload columns (detailed in Table 1). Join key generation begins by sampling $||R_B|| \times \%_{b\_hit}$ keys from a unique set to form build-side hit keys B_{hit}. Next, we sample $sel \times ||R_P||$ keys from B_{hit} using HIT_DIST to create probe-side hit keys P_{hit}. Non-hit keys for both sides are then sampled from distinct key sets B_{miss} (of size $||R_B|| - ||B_{hit}||$) and P_{miss} (of size $||R_P|| - ||P_{hit}||$). Finally, we reorder the combined hit and non-hit keys using B_PAT and P_PAT distributions to produce the final join key columns for both tables. Payload columns are filled with synthetic random data of $type_p$.

5.1.2 Setup. The microbenchmark runs on a machine with Intel®Xeon®Platinum 8474C CPU @ 2.05GHz, 500GB DRAM, 80KB L1 and 2MB L2 cache per core, 97.5MB LLC per socket, and an Intel SSDPF2KE032T1 NVMe SSD (3.2TB, 6.7 GB/s read, 1M 4KB random read IOPS). We run DuckDB v1.1.0 [2] (compiled using `gcc 12.2.0 -O3`) in each experiment to compare the performance of LM (with different fetching algorithms) against DuckDB's native EM.

The tested fetching algorithms are: **naive**, **sort**, **batch** (refer to Section 3.1), and **unique** – a variant of **batch** using a de-duplicate hashmap [7] to batch row IDs. We set $\eta = 0.9$ for **batch** and **unique**. The LM strategy defaults to the **batch** algorithm from Section 5.4. All experiments use a single thread on DuckDB 's uncompressed storage, unless otherwise specified. We enable direct I/O in DuckDB and report the average latency of three trials for each experiment.

5.2 Uniform Distribution

The first row in Figure 9 shows the latency improvement of **batch** LM a build-side attribute over EM where the join keys are randomly distributed. LM becomes advantageous as the `build` size and/or the payload size increases. The observation aligns with our cost model in Section 3 that quantifies the LM/EM difference as $Fetch(EM) + Mem(EM) - (Fetch(LM) + Mem(LM))$. The hash table construction cost $Mem(EM) - Mem(LM)$ is proportional to $||R_B|| * (p - 4)$ where p is the payload size, and row IDs are 4 bytes. For variable-length

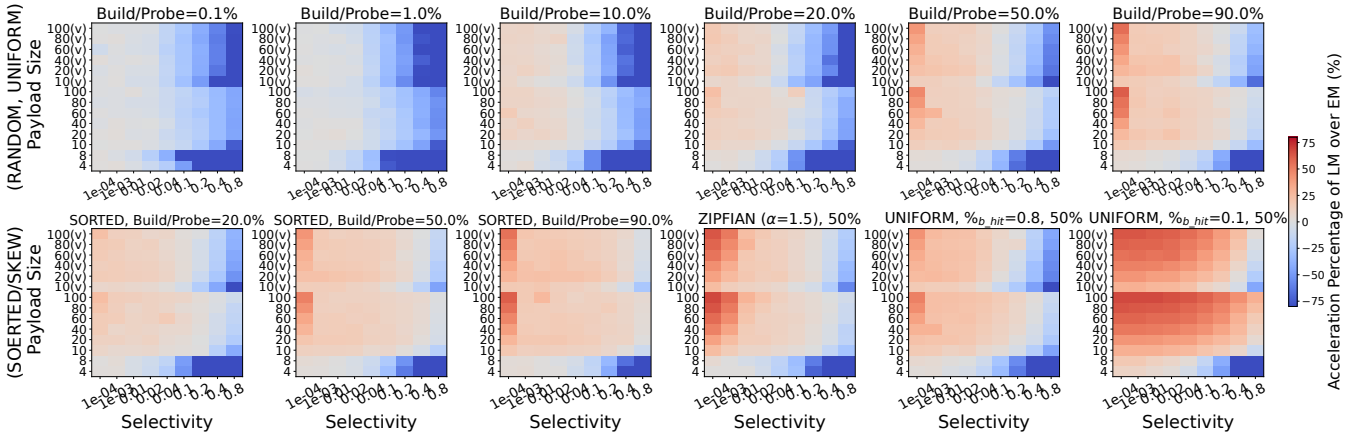
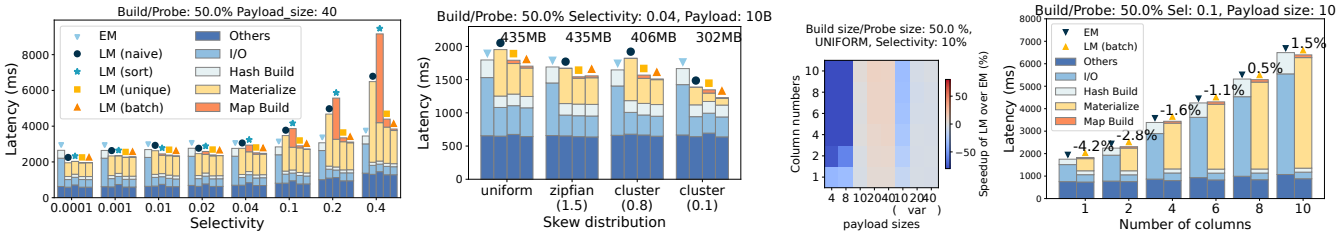


Figure 9: Performance Heatmap – Each subfigure represents a different build size with varying join selectivity and payload size. The first row displays results with HIT_DIST = Uniform and B_PAT = Random, yielding fully random result row IDs. The first three subfigures in the second row depict scenarios where both sides contain sorted row IDs, while the remaining subfigures illustrate cases with skewed hit keys. Note that y-axis labels with suffix "(v)" denotes variable-length string payloads.



(a) Different selectivity **(b) Data skew** **(c) Multi-col heatmap.** **(d) Multi-columns Breakdown.**

Figure 10: Latency Breakdown – I/O and Hash Build represent Table Scan or Hash Build latency; Materialize and Map Build record the time of the Fetch or Batch phase of each fetching algorithm; Others reports all other overheads.

strings, LM's benefit is reduced because of an additional binary search across Column Segments within a Row Group per fetch.

Moreover, LM's advantages increase as selectivity decreases. Figure 10a shows that LM reduces I/O and Hash Build time but increases Map Build and Materialize time. I/O time during materialization is not reported because DuckDB's FetchRow API tightly couples I/O and computation. While Materialize time for all LM strategies grows with selectivity, **batch** and **unique** exhibits the most gradual increase and outperforms EM at selectivities below 10%. I/O remains nearly constant for both LM and EM due to randomly distributed join keys, allowing few block skips except at very low selectivity. The **unique** method's de-duplication offers limited benefit (as row IDs are mostly distinct) while increasing Map Build overhead at higher selectivity. The **sort** strategy underperforms **naive** when selectivity is high, as sorting costs outweigh the marginal benefits(when Row Groups fit into the L2 cache).

The first three subplots in the second row of Figure 9 depict scenarios with sorted result row IDs, which leads to a sequential fetching pattern. Compared to the random case above, LM exhibits small performance improvement because the **batch** algorithm already preserves locality effectively even with random row IDs.

5.3 Data Skew

Figure 9 (2nd row, last three subplots) illustrates the scenarios with skewed result row IDs. The size ratio `build` / `probe` = 50%. LM achieves a greater performance gain over EM than in the random case (Figure 9, 1st row, subplot 5). Figure 10b shows a breakdown with total I/O marked per bar group. When the hit keys follow a Zipf distribution (HIT_DIST=Zipfian) where the result row IDs contain duplicates or heavy hitters, the total I/O is largely unchanged while Materialize time slightly improves (≈ 6%) because most row IDs in each group are identical. **unique** outperforms **batch** by eliminating duplicate row ID accesses, thus reducing Materialize time. For clustered build-side hit keys (B_PAT=Clustered), only $\%_{b\_hit}$ proportion of Column Segments contain hit keys, thus avoiding I/Os to irrelevant Segments during value fetching. Decreasing the cluster ratio reduces total I/O, thereby lowering Materialize time.

5.4 Multiple Columns

This section studies late materializing (**batch**) multiple attributes from a build-side table configured with 2, 4, 6, 8, 10 payload columns (of the same size for simplicity). Figure 10c reports the performance when $\|R_B\| = 50\%\|R_P\|$. Figure 10d highlights LM's increasing

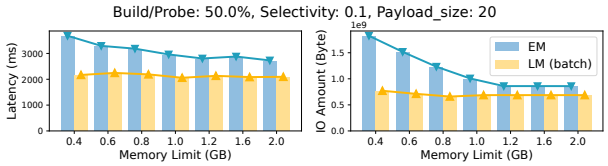


Figure 11: Latency and I/O amount under limited memory.

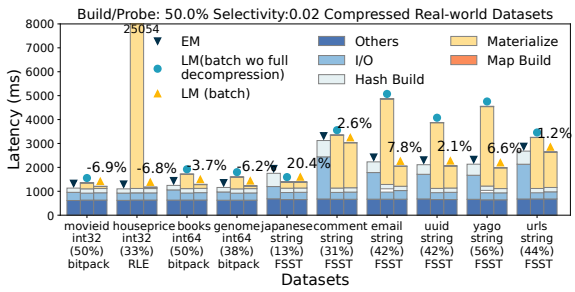


Figure 12: Compressed Breakdown – x-axis shows payload name, type, build-side table's compression ratio, and compression technique. Latency reduction of **batch** LM over EM is marked.

speedup over EM as the number of columns grows, because the attributes can share the same row ID column. The approximate Materialization Cost for EM and LM are $(\|R_B\| \times \#\text{col} \times p) + (\#\text{col} \times T_S)$ and $(\|R_B\| \times 4) + (T_B + \#\text{col} \times T_F)$, respectively. LM's Fetching Cost includes only a single T_B term because all attributes share one Batch Phase (as in Equation (2)). $\frac{Mat(EM)}{Mat(LM)}$ increases with the number of columns, indicating LM becomes relatively faster.

5.5 Limited Memory

We evaluate LM against EM under memory constraints using DuckDB's `memory_limit` parameter. DuckDB spills data to disk upon a buffer-pool overflow. Figure 11 shows query latency and total I/O volume under varying memory constraints. As the memory limit is reduced, EM exhibits a significant increase in both latency and I/O volume. In contrast, LM avoids loading the payload column into memory during Table Scan and reduces the hash table size by replacing payloads with Row IDs. LM achieves a more efficient memory footprint, advantageous as memory becomes more critical [13, 21, 22].

5.6 Compressed storage

We evaluate the performance of LM and EM on compressed storage by populating the payload column with real-world datasets [4, 5, 17, 27]. LM's performance gain over EM is smaller on most string datasets compared to uncompressed storage. First, compression alters element counts in each Column Segment, forcing LM to perform binary searches to locate elements. Second, EM's initial scan cost is offset by reduced I/O and low sequential decompression overhead. Third, point accessing compressed elements during fetching in LM is computationally costly for RLE or FSST. Our full_decompression optimization, which decompresses entire Column Segments upon first access, significantly boosts LM's performance.

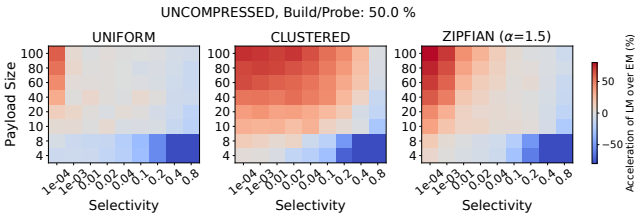


Figure 13: Performance heatmap of Probe side attributes

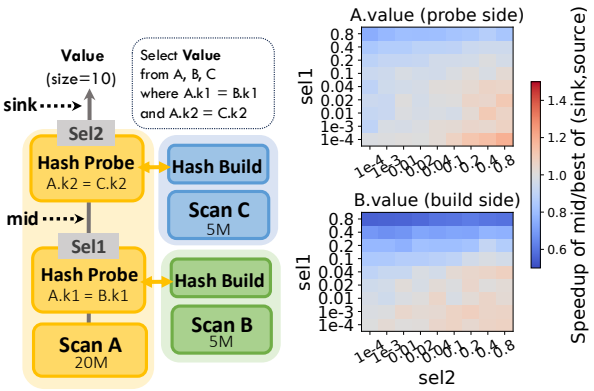


Figure 14: Mid-pipeline late materialization of 2 Join queries.

5.7 Probe Side

For late materializing probe-side payloads, where the obtained row IDs are naturally ordered, as shown in Figure 13, LM improves performance with uniform random keys only at selectivity $= 10^{-4} \sim 10^{-3}$. With clustered (and contiguously located) hit keys, LM achieves substantial performance gains, particularly as selectivity decreases, because only $\%_{p\_hit} = sel * \|R_P\|/256KB$ Column Segments contain the hit keys. LM also performs well when the hit keys follow a Zipfian distribution ($\alpha = 1.5$, skewing heavily toward a few distinct keys, and identical keys are located together).

The experimental results stem from two cost components. First, the Memory Copy Costs are identical for EM and LM because the probe-side payloads are not involved in any row-oriented data structure operations. Second, the Fetching Cost $T_S(N, \#S)$ is dominated by the number of loaded Column Segments. At selectivity $\approx 10^{-4}$ and payload >20B (uniform random), I/O skipping occurs due to #result rows < #Column Segments $= \frac{p * \|R_P\|}{256KB}$ where p is the payload size. This effect is greater for skewed/clustered data, reducing total Materialization Cost.

5.8 Mid-pipeline Materialization

In this section, we investigate the potential benefits of selecting mid-pipeline materialization points. For a set of left-deep, two-join queries with varying selectivity (as shown Figure 14), we compare the latency improvement of materializing an attribute after the first join (i.e., mid-pipeline) against at the pipeline source or sink.

In a vectorized execution engine, mid-pipeline operators must remain stateless and avoid asynchronous operations such as disk

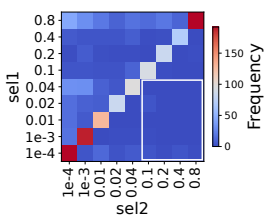


Figure 15: Real middle pipeline selectivity.

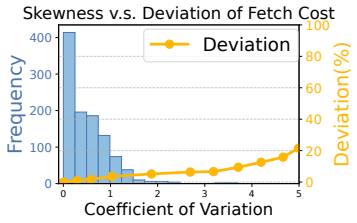


Figure 16: Fetching Cost deviation of different group level skewness.

Method	Pure EM	Pure LM	Optimal	SLM
Total Latency	1757.1	1999.3	1632.9	1651.61

Table 2: Total Latency of the entire workload.

I/O. We introduce a custom pipeline-breaker inserted after the first join to enable mid-pipeline materialization. It splits the original pipeline to allow materialization at the first sub-pipeline's sink using the **batch** strategy. The pipeline-breaker buffers incoming DataChunks from upstream operators and feeds the materialized output to downstream processing. Our evaluation shows that this modification introduces negligible performance overhead (~ 0.5%).

Figure 14 presents the speedups achieved by materializing a 10B attribute in mid-pipeline compared to at the source or sink. Mid-pipeline LM is beneficial when $sel_2/sel_1 \gg 1$ (bottom right corner). We collect the selectivities in all pipelines (> 3 operators) in JOB and TPC-DS. Figure 15 plots the frequency distribution where sel_1 and sel_2 represent the selectivity of a mid-pipeline operator, and at the sink of the corresponding pipeline. We observe that $sel_2/sel_1 \approx 1$ in most real-world pipelines, and it is rare that $sel_2/sel_1 \gg 1$ (the white-box area). This indicates that mid-pipeline materialization is unlikely to offer large benefits for real-world queries.

5.9 Cost Model Efficiency

We verify the effectiveness of our cost model in this section. For scenarios with sorted row IDs, Section 5.7 indicates that LM benefits only from I/O skipping of Column Segments. Assuming randomly distributed join keys, the estimated number of Column Segments accessed in LM is $\#S = min(N, \frac{\|R_P\| * p}{256KB})$ (where N is the cardinality of the materialization point). Our cost model, therefore, chooses the LM strategy when $\#S < N$.

For scenarios with random row IDs (i.e., build-side attributes), we train the Fetching Cost ($\mathcal{B} = 122880$ and $m = \|R_B\|/\mathcal{B}$ according to DuckDB storage) and Memory Copy Cost models using a dataset containing 1134 join groups with varying build-side size $\|R_B\|$, payload size p, and join cardinality N from the microbenchmark. For each join, we evaluate EM and LM on the build-side attribute and measure (1) the attribute fetching latency in EM, (2) the time spent on the Batch / Fetch Phase of LM, and (3) the hash table build time for EM and LM. We use 20% of the data points for training and the rest for testing. We feed the features ($\|R_B\|$, p, and N) of the join under test to the trained models during inference and select the strategy (EM or LM) with the lower predicted cost.

Table 2 presents the latency of running the entire test workload using EM, LM, `Optimal` (via an exhaustive pre-search), or our cost-based `Selective` strategy. Our cost model chooses the correct materialization strategy with 81.1% accuracy and is only 1.2% slower than the `Optimal`.

We next evaluate the robustness of our uniform row ID assumption when training the Fetching Cost model (refer to Section 3.1) against real-world skewness. We generate joins of different selectivities and payload sizes with HIT_DIST=Zipfian under a varying α and use the coefficient of variation (CV, defined as standard deviation/mean) of the row ID counts across all batch groups to quantify the skewness. For each join, we measure the actual Fetching Cost and compute its deviation from the corresponding uniform row ID case. These deviations are plotted in Figure 16, with each CV-level point averaged across selectivities and payload sizes.

Subsequently, we gather real CV statistics from all feasible materialization points (excluding table scans) in JOB and TPC-DS. The histogram in Figure 16 indicates that most cases have CV < 2, where the actual Fetching Cost deviates < 5% from our modeled uniform case. Without prior knowledge of the join key distribution, our model still provides sufficient accuracy for most practical scenarios.

6 END-TO-END EVALUATION

In this section, we demonstrate the performance benefit of Selective Late Materialization by executing queries in JOB [3, 35] and TPC-DS [9] using our modified DuckDB (described in Section 4) with a single thread and uncompressed storage. Each query adopts the join order selected by DuckDB's native optimizer. Notably, varying materialization points do not alter the physical plan structure. We compare SLM[2] against three baselines:

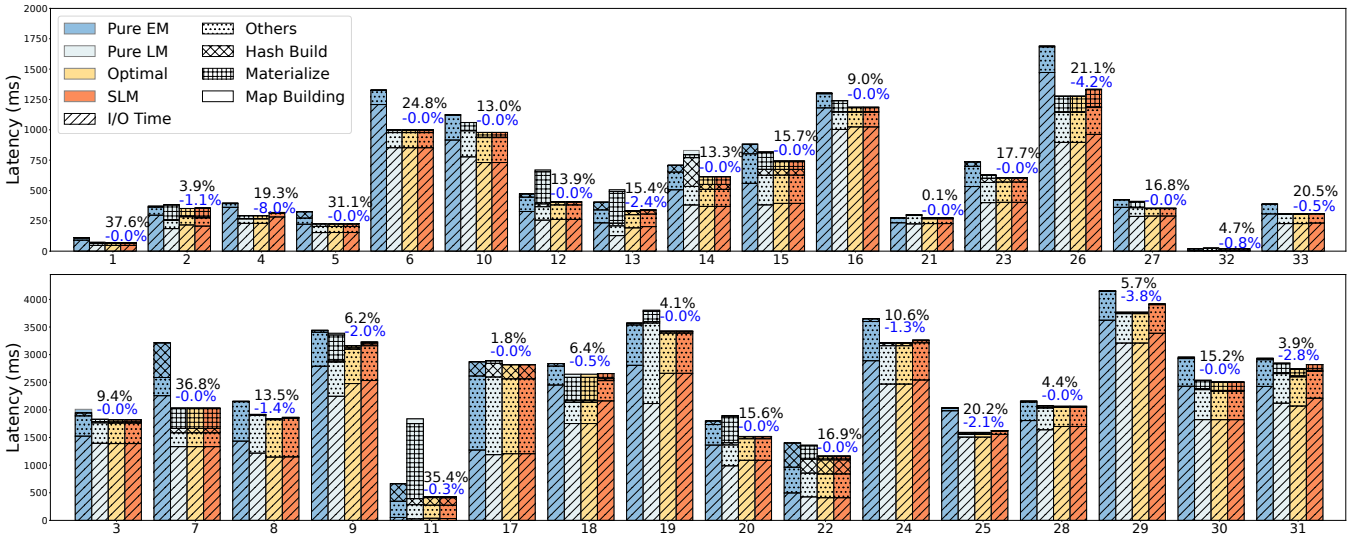
- **Pure EM**: select the *earliest* materialize point for each attribute.
- **Pure LM**: select the *latest* materialization point for each attribute.
- **Optimal**: select the *optimal* materialization point for each attribute via an exhaustive search.

6.1 Benchmark Results

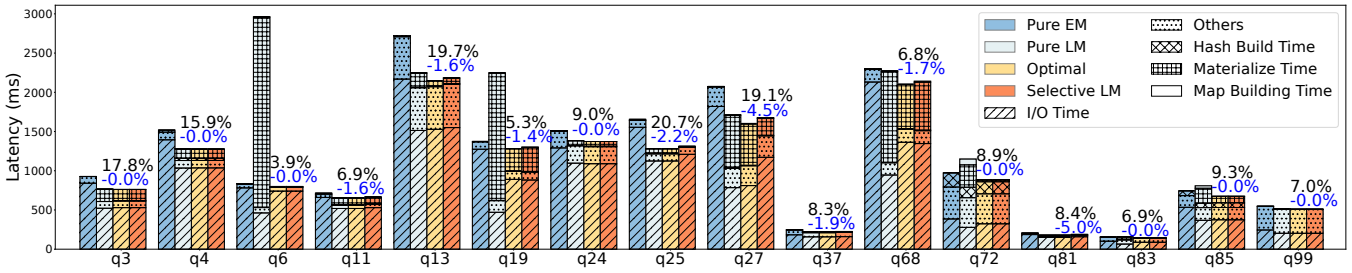
Figure 17a shows the latencies for JOB queries. Pure EM and LM achieve comparable average performance (Pure LM is < 1% faster). However, both strategies lack performance robustness (e.g., Pure EM is 36.8% slower than Optimal in Q7, while Pure LM is 2.8× slower than Optimal in Q11). In contrast, SLM outperforms Pure EM and LM by 14.7% and 8.9% on average, respectively, with a maximum speedup of 76.7%. More importantly, SLM exhibits a robust performance, achieving only a 0.95% average slowdown compared to `Optimal`. The results for TPC-DS are similar. For the 16 selected queries shown in Figure 17b, SLM outperforms Pure EM by 10.9% and Pure LM by 10.3% on average, and is only 1.3% slower than `Optimal`. We found that SLM's performance gains primarily stem from effective I/O skipping and delayed materialization of long `varchar` attributes. We present a few case studies in Section 6.2.

Multi-threading. SLM maintains a consistent ~ 14% performance gain over Pure EM in multi-threaded executions (we tested 4, 8, and 16 threads) of JOB. Figure 18 shows the multi-threaded latencies

[2]For a `varchar` attribute, we feed its average length to the cost model as its size.



(a) JOB result – We divided the query latency of each query template into two sub-figures based on query latency (whether ≤ 2000 ms).



(b) TPCDS result – Scale factor = 10.

Figure 17: Public benchmark result – Black numbers show SLM speedup over Pure EM; blue numbers show SLM degradation vs Optimal.

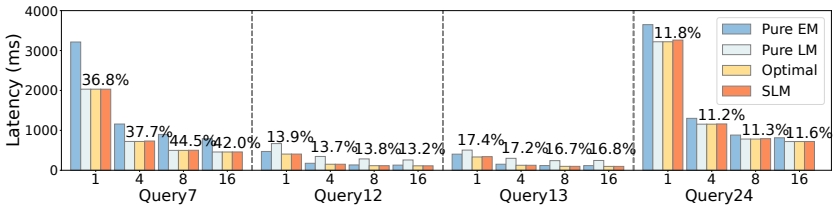


Figure 18: Multi-threading – We present the query latency with 1,4,8,16 threads.

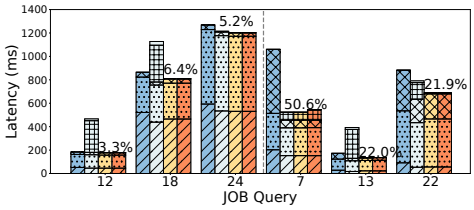


Figure 19: In memory JOB query latency.

of four representative JOB queries. We found that SLM achieves a higher latency reduction over Pure EM in Query 7 as we increase the number of threads. This is because the hash build time accounts for a significant portion of Query 7's runtime, and hash table construction scales worse than the I/O operations due to intensive memory copies and random accesses. Since SLM builds more lightweight hash tables, it scales better than Pure EM overall.

In-memory Scenarios. We configure DuckDB's buffer pool to 40GB (sufficient for all JOB tables and intermediate results) and preload all relevant tables into memory before execution. We rerun the JOB benchmark and observe that SLM achieves an average latency reduction of 13.2% and 14.8% against Pure EM and LM, respectively. Figure 19 details the runtime breakdowns for 6 selected queries, where "I/O time" refers to filling DataChunks with table data from memory only. SLM's improvement over Pure EM diminishes for the left three queries because SLM loses one of its primary advantages in reducing disk I/O. For queries such as the right three in Figure 19, however, SLM's improvement grows because its reduction in the hash build time becomes more impactful with smaller overall query latency due to the elimination of I/O.

6.2 Case Study

We present three case studies in Figure 20 to illustrate why Pure EM and LM are suboptimal in modern analytical databases. Pure EM

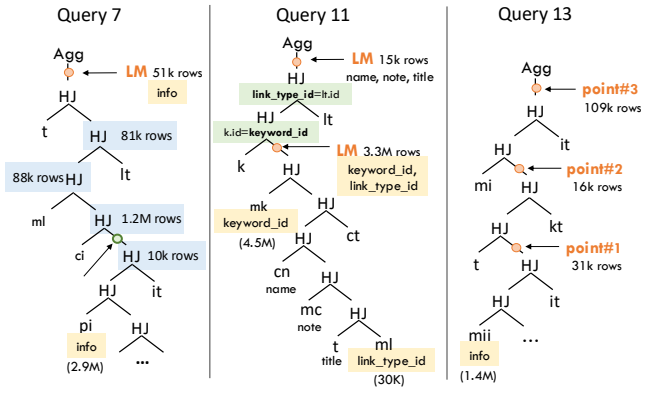


Figure 20: Case study – we annotate the attribute names under discussion and their materialization points with cardinality.

is suboptimal in Query 7 because delaying materialization of the "info" attribute (`varchar` with avg. length = 98B) after the four hash build operators is particularly beneficial. Replacing the "info" values with 4-byte row IDs in these hash tables substantially reduces the Memory Copy Cost (by $\approx (10k + 1.2M + 88k + 81k) * (98 - 4)$ bytes).

Pure LM did not perform well in Query 11 because it mistakenly selects an ultra-late materialization point (cardinality = 3.3M) for two `int32` attributes "keyword_id" and "link_type_id". Late materializing integer attributes does not yield Memory Copy Cost savings, and the cardinality at the chosen materialization point is 0.73× and 111× that of the corresponding base table "mk" and "ml", causes inefficient I/O skipping and high Fetching Cost. In constrast, SLM achieves a 35.6% improvement by only late materializing three string attributes ("name", "note", and "title").

A later materialization point is not always optimal. For example, consider the three LM points (i.e., point #1, #2, #3) for the "info" attribute in Query 13. Although point #3 has lower Memory Copy Cost than point#2 due to two more hash builds between them, its larger cardinality incurs higher Fetching Cost. Our cost model captures this and chooses the intermediate point#2, which is indeed the optimal materialization point.

Additionally, restricting materialization to points with sorted row IDs (and thus forcing EM for all build-side attributes) [16, 31, 40] is often suboptimal because it overlooks potential Memory Copy Cost reductions. In Query 7, for example, such a restriction only considers LM after the 10k hash join (marked in green), yielding an 18.5% latency reduction, far from the 36.8% at the optimal point. Similar suboptimal choices can occur in Query 11 for attributes such as "name", "note", and "title".

6.3 SLM Optimization Overhead

We measure the three components of SLM's optimization overhead (refer to Figure 6) in the above benchmarks:

- SEARCH (0.31-1.03 ms): Identifies feasible materialization points; its duration depends on query complexity (# pipelines).
- INFERENCE (≈ 2.5 ms): Collects features for each attribute at all feasible materialization points and loads our trained model (using scikit-learn in Python) to batch process all features and inference costs.
- OPTIMIZE (0.12-0.72 ms): Re-optimizes the physical plan based on the selected materialization points; its duration depends on the attribute count.

The total optimization overhead is around 3-4 ms, which is minor compared to the execution time of a typical analytical query.

7 RELATED WORK

C-store first introduced Late Materialization [12]. With Hyrise [23], they have explored maximizing I/O skipping by reordering filter execution and dynamically choosing the most efficient filtering strategies. Existing works select materialization strategies coarsely. Some apply either EM or LM holistically to the entire query plan according to resource cost predictions [30]. Vertica [46] and Umbra [16] use LM only for probe-side attributes to avoid the costly out-of-order probing. These methods, however, lack fine-grained, attribute-level EM/LM selection, missing optimization opportunities. PosDB pioneered hybrid EM/LM within single query plans using "Hybrid blocks" (columnar structures for value lists or row IDs) [20, 28]. However, it only evaluates manually crafted plans for two queries, lacking systematic modeling and generalization.

Other works addressed the out-of-order probing problem of the right-side table after joins. Jive-Join [36] and FlashJoin [48] fully sort the row IDs to enable single-scan fetching but have to reorder the output values with an additional sort. Radix Decluster [38] uses cache-sized groups, a cache-conscious radix sort in its Batch Phase, and a dual algorithm in its Output Phase. Rare-Join [45] groups row IDs by disk page and loads the maximum number of groups fit in memory in the Fetch Phase, ignoring the cache efficiency. These fetching strategies, however, overlook modern database features and incur noticeable overhead in their Batch and Output Phases. They are also limited to single joins that eagerly materialize all payloads strictly after each join.

Two future directions could extend Selective Late Materialization (SLM). First, Sideways Information Passing (SIP) aims to prune data early in scans by transferring filters (e.g., bloom filters) between join relations [19, 29, 43, 49]. Current practice [46] often limits SIP filter construction to the build-side for probe-side pruning. Combining this with SLM could yield further performance improvements because LM build-side payloads is beneficial, as shown in Section 6.1. Second, as analyzed in Section 2.2, deferring payload materialization operators with row-oriented data structures reduces the Memory Copy Cost. While this paper focused on hash build operators, extending this to other operators such as Top-K could also yield performance benefits.

8 CONCLUSION

This work revisits the concept of late materialization in modern vectorized execution engines, offering a detailed analysis of its trade-offs. Considering modern hardware performance characteristics, we introduce the problem Selective Late Materialization, which involves attribute-wise intelligent selection of materialization points. We propose a cost-based solution and validate its effectiveness through microbenchmarks and real-world query evaluations.

REFERENCES

[1] 2022. Apache Parquet. https://parquet.apache.org/.

[2] 2024. Duckdb v1.1.0. https://github.com/duckdb/duckdb/releases/tag/v1.1.0.

[3] 2024. JOB implement. https://github.com/gregrahn/join-order-benchmark?tab=readme-ov-file.

[4] 2024. Kaggle Movie ID dataset. https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset?select=rating.csv.

[5] 2024. Kaggle USA Real Estate Dataset. https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset?select=realtor-dataset-100k.csv.

[6] 2025. Apache Doris. https://github.com/apache/doris.

[7] 2025. emhash. https://github.com/ktprime/emhash.

[8] 2025. StarRocks. https://github.com/StarRocks/starrocks.

[9] 2025. TPC-DS Benchmark Standard Specification. https://www.tpc.org/tpcds/.

[10] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[11] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.

[12] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. 2006. Materialization strategies in a column-oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 466–475.

[13] Minseon Ahn, Thomas Willhalm, Norman May, Donghun Lee, Suprasad Mutalik Desai, Daniel Booss, Jungmin Kim, Navneet Singh, Daniel Ritter, and Oliver Rebholz. 2024. An Examination of CXL Memory Use Cases for In-Memory Database Management Systems using SAP HANA. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3827–3840.

[14] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[15] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[16] 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 the 2021 International Conference on Management of Data*. 168–180.

[17] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[18] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[19] Kjell Bratbergsengen. 1984. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*. 323–333.

[20] George A Chernishev, Viacheslav Galaktionov, Valentin V Grigorev, Evgeniy Klyuchikov, and Kirill Smirnov. 2022. A Comprehensive Study of Late Materialization Strategies for a Disk-Based Column-Store.. In *DOLAP*. 21–30.

[21] Yannis Chronis, Anastasia Ailamaki, Lawrence Benson, Helena Caminal, Jana Giceva, Dave Patterson, Eric Sedlar, and Lisa Wu Wills. 2025. Databases in the Era of Memory-Centric Computing. In *Cidr*, Vol. 25.

[22] Manos Frouzakis, Juan Gómez-Luna, Geraldo F Oliveira, Mohammad Sadrosadati, and Onur Mutlu. 2025. PIMDAL: Mitigating the Memory Bottleneck in Data Analytics using a Real Processing-in-Memory System. *arXiv preprint arXiv:2504.01948* (2025).

[23] Martin Grund, Jens Krueger, Matthias Kleine, Alexander Zeier, and Hasso Plattner. 2011. Optimal query operator materialization strategy for hybrid databases. In *Proceedings of the 2011 Third International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA'11)*. 169–174.

[24] Alan Halverson, Jennifer L Beckmann, Jeffrey F Naughton, and David J Dewitt. 2006. *A comparison of c-store and row-store in a common framework*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[25] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. 2006. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd international conference on Very large data bases*. Citeseer, 487–498.

[26] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 297–308.

[27] A. Kipf, R Marcus, A Van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. 2019. SOSD: A Benchmark for Learned Indexes. (2019).

[28] Evgeniy Klyuchikov, Michael Polyntsov, Anton Chizhov, Elena Mikhailova, and George Chernishev. 2024. Hybrid Materialization in a Disk-Based Column-Store. In *Proceedings of the 7th Joint International Conference on Data Science & Management of Data (11th ACM IKDD CODS and 29th COMAD)*. 164–172.

[29] Paraschos Koutris. 2011. Bloom filters in distributed query execution. *University of Washington, CSE* 544 (2011).

[30] Chi Ku, Yanchen Liu, Masood Mortazavi, Fang Cao, Mengmeng Chen, and Guangyu Shi. 2014. Optimization strategies for column materialization in parallel execution of queries. In *Database and Expert Systems Applications: 25th International Conference, DEXA 2014, Munich, Germany, September 1-4, 2014. Proceedings, Part II 25*. Springer, 191–198.

[31] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).

[32] 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 *Companion of the 2024 International Conference on Management of Data*. 5–17.

[33] 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 the 2016 International Conference on Management of Data*. 311–326.

[34] 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 the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[36] Zhe Li and Kenneth A Ross. 1999. Fast joins using join indices. *The VLDB Journal* 8 (1999), 1–24.

[37] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–28.

[38] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. 2004. Cache-conscious radix-decluster projections. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 684–695.

[39] Stratos Idreos Fabian Groffen Niels Nes and Stefan Manegold Sjoerd Mullender Martin Kersten. 2012. MonetDB: Two decades of research in column-oriented database architectures. *Data Engineering* 40 (2012).

[40] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[41] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[42] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[43] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. 2009. Optimizing distributed joins with bloom filters. In *Distributed Computing and Internet Technology: 5th International Conference, ICDCIT 2008 New Delhi, India, December 10-12, 2008. Proceedings 5*. Springer, 145–156.

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

[45] Mehul A Shah, Stavros Harizopoulos, Janet L Wiener, and Goetz Graefe. 2008. Fast scans and joins using flash drives. In *Proceedings of the 4th international workshop on Data management on new hardware*. 17–24.

[46] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization strategies in the vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1196–1207.

[47] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[48] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A Shah, Janet L Wiener, and Goetz Graefe. 2009. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 59–72.

[49] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M Patel. 2017. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *Proceedings of the VLDB Endowment* 10, 8 (2017), 889–900.

[50] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1349–1350.