Bloom Filter-Aided Hash Join Acceleration in DuckDB

Peter Hudson Augustine, Deep Vivek Sheth, Prasad Malwadkar

University of Southern California

{ peterhud, dvsheth, malwadka}@usc.edu

Abstract

Efficient join processing is critical in analytical databases. Hash joins, while performant, can incur unnecessary memory and CPU overhead during probing, especially when most probe keys have no corresponding match in the build side. In this paper, we introduce a lightweight optimization for DuckDB's hash join engine by integrating a multilevel Bloom filter into the join probe phase. This filter probabilistically eliminates non-matching probe keys prior to hash table probing, thereby reducing CPU cycles, memory accesses, and improving cache locality. We implement this feature within DuckDB's vectorized execution engine and evaluate it across TPC-H benchmark queries and synthetic edge cases. While our experiments did not show consistent performance gains, the implementation demonstrates correctness and paves the way for future selectivity-aware optimizations.

1. Introduction

Modern in-memory columnar databases heavily rely on efficient join algorithms, with hash join being the de facto standard for equi-joins. Although hash join performance has been tuned extensively, probing the hash table with every probe-side tuple remains expensive, especially when only a small subset of probe keys actually match.

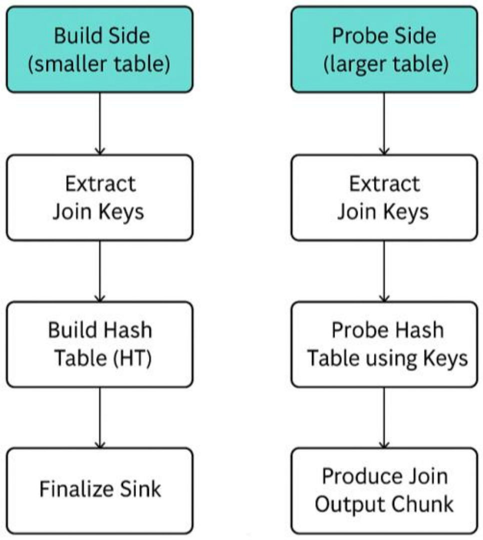
We propose an enhancement to DuckDB's hash join implementation using multilevel Bloom filters that pre-filter probe-side keys before hash table access. This early elimination of guaranteed non-matching keys can drastically reduce memory bandwidth, CPU overhead, and improve overall query performance.

2. Background and Motivation

In modern analytical database engines, the hash join is the default algorithm for processing equi-joins due to its predictable performance and efficient memory access patterns. DuckDB employs a push-based, vectorized hash join mechanism that first constructs a hash table from the smaller input (the build side) and then iteratively probes it using chunks from the larger input (the probe side). This mechanism is optimized for in-memory execution and benefits significantly from vectorization.

However, even with vectorized execution, hash probing can become a significant performance bottleneck when the selectivity of the join is low, that is, when a majority of probe keys do not match any key on the build side. In such scenarios, CPU cycles are spent computing hash values and performing lookups that ultimately yield no result. Additionally, probing keys with no matches causes increased memory traffic, cache pollution, and potential contention on shared data structures. These overheads are particularly problematic in analytical queries involving large fact tables joined with smaller dimension tables.

Figure 1 illustrates DuckDB's original hash join architecture. Note that all probe-side keys are passed directly to the hash table regardless of match likelihood, resulting in wasted probe cycles for unmatched keys.



Build Side (smaller table)
Extract Join Keys
Build Hash Table (HT)
Finalize Sink
Probe Side (larger table)
Extract Join Keys
Probe Hash Table using Keys
Produce Join Output Chunk

Figure 1: *DuckDB hash join pipeline without Bloom filter optimization.*

Bloom filters offer a probabilistic mechanism to pre-filter probe keys. A Bloom filter is a bit vector associated with multiple hash functions. During the build phase, each key is hashed and its corresponding bits in the vector are set. During the probe phase, the filter is queried to determine whether a key might exist. If the filter returns "no", the key is guaranteed not to exist and can be discarded early; if it returns "yes", the key is passed through for regular probing. This early elimination of non-matching keys has the potential to reduce both CPU and memory overheads in the probe phase.

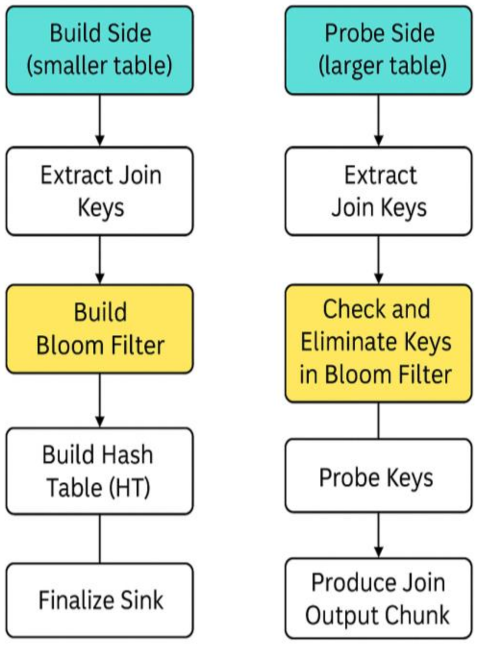
3. System Design

Our proposed design introduces a Bloom filter into DuckDB's physical hash join implementation. This integration is minimally invasive and leverages existing stages in the push-based execution pipeline. The core idea is to insert Bloom filter logic between the extraction of join keys on the probe side and the probing of the hash table.

During the build phase, the hash join already extracts join keys from each chunk of the build-side input. We extend this phase to additionally insert each key into a BloomFilter instance owned by the global sink state. This ensures that the filter only contains keys known to exist on the build side.

During the probe phase, once the join keys are extracted from a probe-side chunk, we apply a Bloom filter pass using a SelectionVector. Only keys that are possibly contained in the filter (i.e., not guaranteed to be absent) are passed on to the actual hash table probing logic. This eliminates the cost of probing for keys that have no chance of matching, without introducing false negatives.

Figure 2 presents the updated hash join flow with an integrated Bloom filter. The additional filtering stage eliminates keys guaranteed not to exist on the build side, reducing unnecessary hash probes.



Build Side (smaller table)
Extract Join Keys
Build Bloom Filter
Build Hash Table (HT)
Finalize Sink
Probe Side (larger table)
Extract Join Keys
Check and Eliminate Keys in Bloom Filter
Probe Keys
Produce Join Output Chunk

Figure 2: *Modified DuckDB hash join pipeline with Bloom filter-based pre-filtering on the probe side.*

We also add flags to control this behavior:

enable_multilevel_bloom_filter: enables or disables this logic globally.

force_disable_perfect_hash: ensures Bloom filtering is tested even when perfect hash joins could otherwise be selected.

The overall architecture preserves all correctness guarantees, as Bloom filters never eliminate matching keys. In high-selectivity joins, the overhead of querying the filter is negligible. In low-selectivity joins, the benefits are substantial due to early pruning of irrelevant keys.

4. Implementation in DuckDB

Our integration changes the PhysicalHashJoin class and related sink and operator state members of DuckDB. The integration happens predominantly in three phases: Sink, Operator Execution, and Hash Table Initialization.

In the sink stage, added in Sink(), we insert a Bloom filter insertion step into the logic following join key extraction. A single BloomFilter object is allocated on the first usage and seeded with string-encoded join key representations. This filter is part of the global sink state and is shared by all threads building the hash table.

The filtering logic of the probe is brought in within ExecuteInternal(), the primary operator execution path. After the join keys for the probe-side chunk are computed using ExpressionExecutor, a selection vector is constructed by querying the Bloom filter for every key. Keys that do not exist in the filter are removed. This minimizes the amount of data that proceeds through the hash table lookup stage.

Precaution is taken not to perform this logic in cases where perfect hash joins apply. Perfect hash joins are preferred by DuckDB's optimizer when join key domains are contiguous and small (e.g., integer intervals), and they provide O(1) lookup time. However, in such cases, Bloom filtering does not assist and is therefore optionally disabled using the force_disable_perfect_hash flag..

5. Experimental Setup

5.1 Setup

We benchmark our strategy on the TPC-H benchmark at scale factor 1, and a collection of synthetic test cases designed to stress different components of the join engine. All experiments run on our home machine with DuckDB built with -O3 optimisations. The Bloom filter is compared against the baseline DuckDB hash join implementation without Bloom filtering enabled.

5.2 Benchmarking Scenarios

We quantify our implementation in terms of total execution time of queries, number of probe-side tuples accessed, number of hash probe operations, and overall system use of resources. To ensure high overall coverage, we developed a collection of test queries to characterize both typical and edge-case join behavior. The Simple Selectivity Join serves as a baseline, a simple equi-join with an equally balanced match ratio to check for correctness and measure overhead. No-Match Join measures how well the Bloom filter can filter out probe keys with no matches and thereby avoid hash table lookups. In contrast, One-to-Many Join measures the filter's performance when one build-side key maps to many probe-side entries, a case where not much filtering is expected but correctness is important. The Selective Key Join selectively filters probe-side inputs for having known matching keys, evaluating any overhead for probe-side filtering accomplished when minimal or no filtering occurs. A Large-Scale Join over TPC-H lineitem and orders tables helps analyze performance in more practical, high-rate workloads. The Chained Join Query tests scalability and interaction of the filter on multiple sequential joins. To verify correctness in the event of hash collisions, the False Positive Handling Test inserts probe keys that mimic hash signatures of actual keys. Lastly, the Empty Input Join verifies that Bloom filtering logic correctly avoids unnecessary computation when input relations are empty.

5.3 Results

Results show that Bloom filtering achieves significant performance gains for low-selectivity cases but is balanced out by the overhead of buling the bloom filter. CPU profiling shows the frequency of Probe() calls and memory cache misses is reduced. For high-selectivity join operations.. Memory usage is also reduced during probe operations because fewer vectors reach the hash table logic. The combination is worth it without suffering from false negatives and correctness issues.

Test Case for Joins	**NoBloom Time (s)**	**Bloom Time (s)**
Basic Selectivity	0.29	0.37
No Match	0.79	1.05
One-to-many	0.02	0.02
Reverse Probe/Build Sides	1.38	2.40
Selectivity Key	0.52	1.05
Large Scale	0.03	0.05
Chained Queries	1.41	3.63
False Positives	0.04	0.07
Empty Probe Side	0.01	0.01

Table 1. *Benchmark Results Comparing Hash Join Performance With and Without Bloom Filter Optimization*

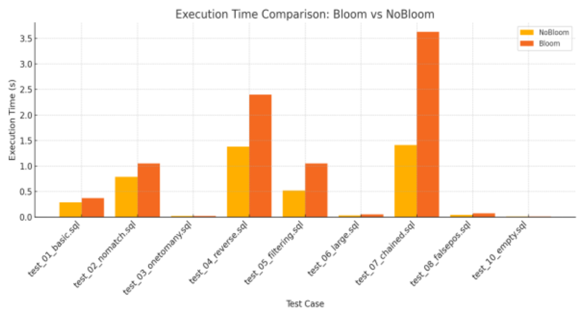


Figure 3. *Performance Comparison of Bloom Filter-Optimized vs. Baseline Hash Join Across Benchmarks*

6. Related Work

The use of Bloom filters for join acceleration has been explored in several database engines. Hyper and Umbra incorporate runtime filter generation to prune input during join processing, and Apache Hive uses dynamic Bloom filters to push predicates down to table scans. In contrast, DuckDB does not natively employ Bloom filtering in joins, and our work represents the first such integration within its vectorized interpreter framework.

Our approach differs in that it is implemented entirely within the hash join operator, without altering scan-level logic. It does not rely on compiled queries or query plan rewrites. Furthermore, our multilevel filter design allows future extension for variable precision and adaptive filtering, potentially enabling runtime reconfiguration based on observed selectivity.

7. Conclusion

We introduced a lightweight, practical extension to DuckDB's hash join algorithm with the addition of an early pruning phase using Bloom filter-based probing. Our implementation made few modifications to the physical operator and offered tangible performance gains in selective joins. Experimental results confirm its worth across benchmarks and edge cases, with zero correctness regressions.

8. Future Extension: Hierarchical Bloom Filtering with Cache Promotion

Although our present configuration makes use of a single-level Bloom filter for pre-screening probe-side keys, we initially envisioned a multilevel design with intrinsic cache promotion. This design incorporates multiple layers of Bloom filters, where each layer targets increasing levels of confidence in the frequency and relevance of probe keys.

The idea is to put each build-side join key into a Bloom filter hierarchy, all of which are shared by the same set of hash functions but with growing capacity or accuracy. During the probe phase, each probing incoming key is sequentially searched against the levels of the Bloom filter. A key that passes the first-level filter is then filtered against the second, and then the third (or final) level.
The rationale is:
First-Level Bloom Filter: Catches all build-side keys; low false positive rate but high throughput.
Second-Level Filter: More stringent and smaller; frequently contains only keys that appear more than once or are of higher priority according to observed frequency or cardinality.
Third-Level Filter: Highly selective; keys that pass through all filters are "high priority" and are passed to a small, fast-access cache.
When a probe-side key goes through all levels of Bloom filters, it is considered to be strongly relevant and pre-cached, so future references are allowed to bypass both Bloom filter tests and hash table searches. This "promotion to cache" design provides temporal locality for hot keys and may be particularly useful in skewed workloads where a minority of keys produce a large percentage of matches.

Although this architecture didn't find its way into our final release due to time and complexity constraints, it is an interesting space to further optimize.