Adaptive Factorization Using Linear-Chained Hash Tables

Paul Groß*
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
gross@cwi.nl

Daniel ten Wolde*
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
dljtw@cwi.nl

Peter Boncz
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
boncz@cwi.nl

ABSTRACT

We introduce factorized aggregations and worst-case optimal joins in DuckDB with an adaptive mechanism that only uses them when they enhance query performance. This builds on the adoption of a new hash table design ("Linear-Chained") for equi-joins. Our first insight is that the collision-free chains of this new design enable efficient factorized and worst-case optimal processing. We further defer the decision to use factorization and worst-case optimal joins from optimization to runtime. Our second insight is that we can obtain accurate statistics, even if the join inputs lack these (e.g. because they are sub-queries or Parquet files), by leveraging runtime heuristics and constructing efficient *on-the-fly sketches*, during the hash join build. Finally, we show that machine learning models using these metrics can achieve close to optimal performance with a high accuracy. Furthermore, we propose heuristic-based approaches that offer comparable performance to these models, while relying on cheaper to obtain run-time statistics and being more explainable.

1 INTRODUCTION

A large body of research on worst-case-optimal joins (WCOJ) [19] and factorization [20] has identified algorithms that reduce the complexity class of certain queries, compared to classical methods. However, factorized techniques so far have not made it into mainstream systems, and where WCOJs have been deployed, they still face query optimization challenges. They typically require upfront investment at least proportional to the query input – such as sorting or the creation of a specialized data structure like tries – and subsequently also involve more complex logic. Therefore, faster results are typically only obtained in queries that have intermediate results significantly larger than both the query inputs and output – such that avoiding these intermediates justifies the mentioned upfront investment and complex logic [5]. Hence, the new algorithms cannot *replace* classical ones but become new options for an optimizer to select when opportune. Regrettably, estimating the costs for such queries is hard and requires specialized statistics, as pioneered in graph database systems [7], but whose creation and maintenance on base tables are absent in practice and hard to justify a-priori.

*Both authors contributed equally to this research.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2025. 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

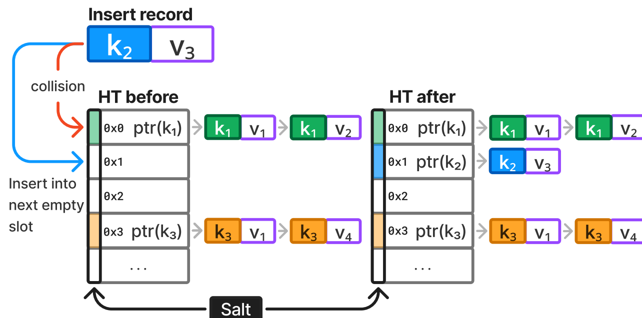

Figure 1: The Linear-Chained Hash Table combines linear probing and chaining. Inserting k_2 leads to a hash collision with k_1. Therefore, k_2 is inserted into the next available bucket. The chains consist of records with the same key.

In this paper, we break this dilemma by making factorization and WCOJs *adaptive*, making the following contributions:

- the "Linear-Chained" hash table design that uses linear probing to handle key collisions and chains for duplicate keys.
- an implementation and evaluation of factorized aggregation and WCOJs in DuckDB [21], exploiting these hash tables.
- a mechanism to postpone the decision to use these algorithms from query optimization to runtime by putting statistics from sketches created during hash join builds into a simple model with high accuracy.

Background. DuckDB [21] is an embeddable analytical database system originally developed at CWI. It has a push-based vectorized execution engine [6] with morsel-driven multi-core parallelism [15]. DuckDB uses compressed columnar storage with MinMax-powered predicate pushdown [1] and keeps HyperLogLog (HLL) statistics on all columns for join cardinality estimation [10]. There is support for out-of-memory data processing for many of its operators [14]. Finally, it has a feature-rich extension framework that can be used to add parser and optimizer rules and introduce new operators or UDFs [24]. One such extension is DuckPGQ [24] which adds SQL/PGQ support for graph processing from SQL:2023. DuckPGQ translates graph pattern matching queries into relational query plans with multiple *n:m* joins. This can trigger the WCOJs and factorized aggregation presented here. However, our work targets *all* SQL queries in DuckDB, not only those phrased in SQL/PGQ.

Classical systems perform joins table-at-a-time in some table join order. In contrast, WCOJs, such as the Leapfrog Triejoin [25], are algorithms that process a whole join graph in one go. They order the join attributes, and for each attribute value, find a list of tuple bindings. They then intersect these lists for all attributes in the chosen order, using data structures that ensure that the runtime of these intersections is in the order of the shortest list (this

typically involves a trie [9, 19, 25]). Note that WCOJs still generate redundancies. Just generating the query result is by definition worst-case optimal, but that result can be redundant. For instance, in a social network multi-step friendship query, all further join bindings from one friend onwards are the same for all people who have that friend. Factorized Query Processing [3] aims to compactly represent query results to avoid such redundancies, using compact representations. These are structures that compress results using algebraically [11, 17, 20]. For instance, if we have a friends relation from user A to users B, C, and D, we can represent this through *flat* tuples as $\langle A\rangle \times \langle B\rangle \vee \langle A\rangle \times \langle C\rangle \vee \langle A\rangle \times \langle D\rangle$. The corresponding *factorized* representation $\langle A\rangle \times (\langle B\rangle \vee \langle C\rangle \vee \langle D\rangle)$ is more sparse.

We build our factorized processing on the idea of the 3D Hash join [8] that used *hierarchical* chaining. For each bucket, there is first a chain containing the colliding keys, and for each key, there is then another chain with pointers to the records with that key. This data structure is only efficient for joins with frequent duplicates. Here, we present the Linear-Chained hash table, which is efficient for all equi-joins and enables factorized processing.

Our work is quite close to ongoing research in Umbra [5] that, similar to the 3D Hash Join [8], also splits a join in a *lookup* that delivers a chain pointer and a later *expand* phase. In [5], the authors focus on the query optimization opportunities provided by reordering join-lookup and -expand operators independently, but they also investigate the integration of WCOJ in their factorized framework. Our work differs in that we also look into factorized *aggregations* and investigate how to move the optimization of factorized queries to *runtime*, using on-the-fly computed heuristics and sketches, to avoid the requirement of having detailed base table statistics.

The Kùzu graph database system includes factorized processing and WCOJs [13], but query optimization is still a challenge – we found that it needs optimizer hints to generate a WCOJ even in triangle queries. Graph query optimization over multiple *n:m* joins is an active research topic, e.g., with [7] showing progress in cardinality estimation. But in the general case, estimation errors remain significant [17]. We side-step many of these issues by computing statistics on-the-fly directly on the join inputs.

Outline. In Section 2, we explain the implementation of factorized query processing in DuckDB and its application for queries involving aggregations and joins. Section 3 describes the calculation and exploitation of on-the-fly sketches to make factorization runtime adaptive. Finally, Section 4 contains a performance evaluation, and Section 5 concludes with suggestions for future work.

2 THE LINEAR-CHAINED HASH TABLE

Linear hash tables store records directly in the hash buckets and therefore resolve lookups typically with one CPU-cache miss. While many systems use linear probing for the aggregation operator (GROUP BY), *bucket-chaining* is often used for hash joins. Bucket-chained hash tables store records separately from the hash buckets; so lookups typically generate *two* cache misses: one for the bucket and one for the record. Unlike in aggregation, join hash tables can hold (many) duplicate keys, and then linear hash tables perform worse than chained ones [4]. A second advantage of chaining is the build phase, where the first step is to materialize the records – as for chaining this requires no parallel synchronization, while for linear probing it does. A third advantage is efficient selective joins: by keeping a small bloom-filter (“salt”) in the highest 16 bits of the buckets – which contain 64-bit record pointers – queries that find no match can be identified without accessing the records [15]. Selective workloads therefore do not need to keep the records in the CPU cache, and the buckets are much smaller than the records, so they fit in the caches better than linear hash tables.

In addition, we use the uppermost bit of the salt to mark whether there was a hash collision on a bucket during the build. When probing this bucket, we can now check this *collision bit* to know whether we must do linear probing in case the salt does not match.

As a part of this work, DuckDB v1.1.0 adopted the *Linear-Chained Hash Table*, introduced in this paper. It is a bucket-chained hash table, where in case a newly inserted key collides with an already inserted different key, this is resolved with linear probing, as shown in Figure 1. Our implementation tries the next bucket in case of a collision to profit from cache locality, but other collision handling strategies, such as quadratic probing could also be used.

As a result, chaining is only used when there are multiple records with the same key. Because each bucket stores only records for a single key, the salt is no longer a mini bloom-filter (where each key sets a few bits) but is a 15-bit hash, providing low false positives at a low computational cost.

Compared to DuckDB’s original hash table, this approach has advantages in processing selective joins – thanks to the accurate salt – as well as in joins with many duplicates: after the first probe hit, all subsequent hits on items with the same key can now be generated from the pure chain *without* key comparisons. The disadvantage is the additional key equality check during insertion if a bucket is non-empty; in case this is due to duplicate keys, its probing advantages outweigh this disadvantage – so this only has negative effects for true hash collisions, which are rare given DuckDB’s generous bucket sizing (#buckets ≅ 2x#records)

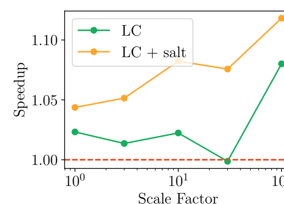

Figure 2: Linear-Chaining (LC) with salt improved TPC-H performance.

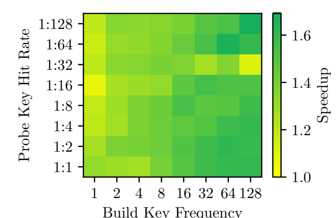

Figure 3: LC is better with selectivity (collision-bit & salt) and build key frequency.

Figure 2 shows the speedup of the linear-chained hash table compared to the previous hash join implementation in DuckDB for the combined runtime of all TPC-H queries. With no salt optimization, the speedup is generally close to the baseline but becomes higher with larger scale factors. Using salt optimization, the new join is faster for every scale factor, with better performance for larger-scale factors. This shows that the salt can reduce the number of cache misses, which becomes more relevant with larger hash tables. The matrix in Figure 3 shows an exhaustive test for different duplicate key frequencies on the build side (X-axis) and probe hit ratios (Y-axis). For the experiments, we chose a probe-side cardinality of 2^{28} and a build-side cardinality of 2^{24}. The matrix shows

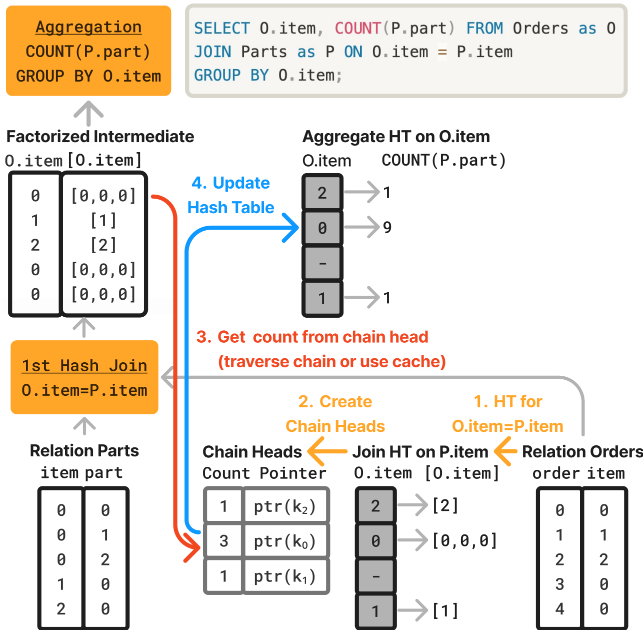

Figure 4: Cached COUNT(expr) aggregate computation: Probing tuples sharing the same pointer can use the chain head's aggregate, avoiding repeated traversal for counting.

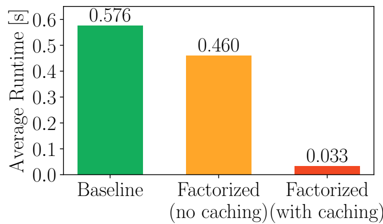

Figure 5: Factorization benefits with and without caching.

that there are no regressions and that the speedup scales with the frequency of the build key, which is due to fewer comparisons due to the key-unique chains. Further, there are also speedups for highly selective joins as they benefit from both the salt comparison and the collision bits. However, queries with high selectivity show lower speedup because many key-unique chains remain unprobed, making the effort to ensure their key uniqueness redundant. Importantly, for our work here, Linear-Chained hash tables provide the basis for factorized processing.

Exploiting Linear-Chained Hash Tables. We can now apply the idea of the 3D Hash join [8] in DuckDB to let a join probe emit *chain pointers* that point to a chain in the hash table rather than expand the result with one result tuple for every hit on the same key. This practical idea is what in factorization theory is called a *d-representation* [17, 20], where factorization happens by pointing to the definition of a result. In contrast to *f-representations*, this approach enables the definition to be reused multiple times by repeatedly emitting the same chain pointer. Both representations avoid join expansion, but the list of values to expand the input with is computed and stored *only once* for d-representations. In contrast, in f-representations, it gets computed and stored potentially multiple times. The 3D Hash join paper used factorization only to *delay* join expansion, such that higher-up filters or selective joins in the same pipeline could eliminate tuples before expansion ("deferred unnesting"). We further pursue this idea in both (grouped) aggregations on *n:m* joins and cyclic hash joins, i.e., in cases where join expansion can be *avoided* and the benefit of factorization is high.

Factorizable aggregates are composable functions like `MIN(expr)`, `MAX(expr)`, `SUM(expr)`, `COUNT(expr)`, whose `GROUP BY` (if present) does not depend on the build side of a not yet expanded join lower in the query pipeline. If the aggregate input `expr` also does not depend on the build side, these do not need to be expanded at all to compute their result; for `SUM` and `COUNT` one just needs to multiply the result with the length of the chain. Further, any composable aggregate whose input `expr` comes from the build side of a factorized join can be (partially) *cached* after computation. The advantage of these cached partial aggregates is that they get computed only *once*, while a key can be probed *many* times, reducing memory access and computation. Once the hash table is built, the count of unique chains is known. We use an array sized to this count, which holds pointers to the chains and space to store partial aggregates, as shown in Figure 4. Instead of directly emitting pointers to the chains, the join probe then emits pointers to this array.

We developed a microbenchmark to evaluate factorization performance for aggregations with and without caching. The benchmark involves two tables: `Orders`, consisting of 500,000 rows containing 10,000 distinct `item` IDs, and `Parts`, which includes 1,000,000 rows with 50,000 distinct `part` IDs and 10,000 distinct `item` IDs.

The query used for this benchmark can be found in the top right corner of Figure 4. DuckDB's non-factorized query plan serves as our baseline. This plan expands the probing tuples due to the duplicates on the build side, resulting in a large intermediate result. As the join and the aggregation are performed on the same key (`o.item`), we can opt for a factorized plan. Using the previously described caching optimization, we only compute the aggregations per key once. We can then cache the results for subsequent accesses, as shown in Figure 4.

`Orders` is the build side as it is smaller. Using the `item` key, with 500,000 entries and 10,000 distinct values, results in an average chain length of 50 in the hash table. This reduces the factorized intermediate size to just 2% of a flat result. For aggregation, each `item` key occurs 100 times on average. When accessing a chain for the first time, we will traverse it to get the chain length and then cache it. Therefore, a full traversal is only needed 1% of the time. Figure 5 shows that even without caching, factorization improves performance, yielding a 1.25x speedup. With caching enabled, we observe a substantial enhancement in runtime with a 17.58x speedup.

Worst-case optimal joins can be especially efficient for cyclic patterns over *n:m* joins, such as shown in Figure 1. Finding cycles within a graph is a common task [22], but challenging from a systems perspective due to the size of the intermediate result being significantly larger than the final result. In DuckDB's standard join query plan, the example triangle query is executed using two (hash) equi-joins: the first join matches the relations `R.dst=S.src` and the

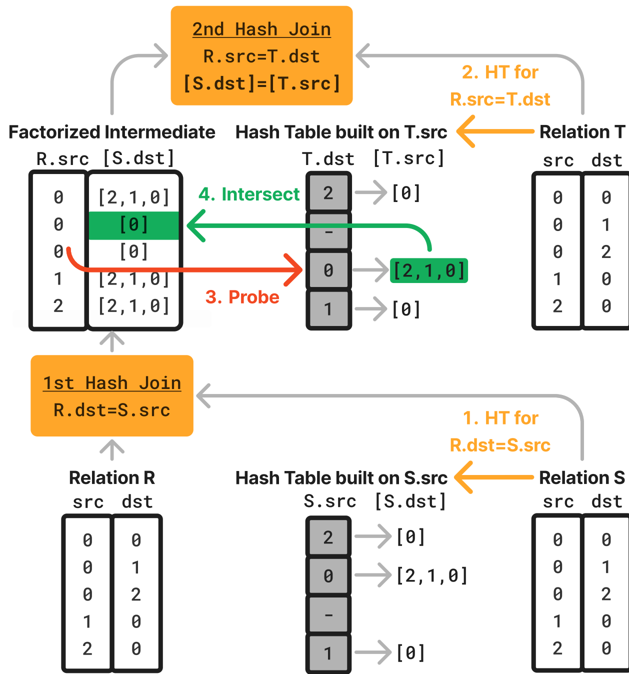

Figure 6: WCOJ plan in DuckDB. The first join probe finds a list (chain pointer). The second join as well. Then, these two lists are intersected to generate the result.

```
# "follows" has edges
# (a "connecting table")
SELECT *
FROM follows R,
     follows S,
     follows T
WHERE R.dst = S.src
  AND S.dst = T.src
  AND T.dst = R.src
```

Listing 1: SQL triangle join.

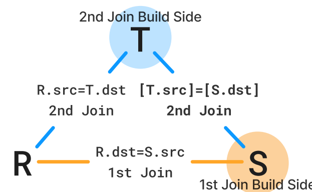

Figure 7: Join Schema.

second closes the cycle with `S.dst=T.src` and `T.dst=R.src`. Specifically, the first hash table is built on table `S` using key `S.src` while the second hash table is built on table `T` using the combined keys `T.src` and `T.dst`.

In contrast, exploiting the Linear-Chained Hash Tables, the first join will not expand but generate a chain pointer representing a list of `S.dst`-s. The second hash table should only be created on `T.dst`. As a consequence, a lookup with `R.src` in the second hash table will find a matching `T.dst` and again deliver a chain pointer, that represents a list of `T.src`-s. The join result can then be computed by *intersecting* the list of `S.dst`-s from the first join with the list of `T.src`-s from the second. Provided the complexity of this intersection is linear to the shortest of the two lists, this is a WCOJ.

This WCOJ query plan is shown in Figure 6. Note that in the second join, we depict the list of `T.src`-s inline in the left join input, though in reality in DuckDB these are chain pointers that point into the first hash table. These chain pointers are a factorized *d-representation* because in the first join, a pointer to the same list of `S.dst`-s is generated for any `R.dst` with the same value.

In the DuckDB implementation of the intersection-join, we use a hash-based approach to allow two lists to be intersected in time linear to the shortest list. For this purpose, we allocate large buffers in which many mini linear hash-tables are stored one after the other, one for each list, each sized by the smallest power of two larger than the list length. The buckets of these mini hash tables are salted pointers to the records already present in the original hash table allocated. Intersection uses the smallest list to probe and performs lookups in the mini hash table of the larger list.

3 FACTORIZING ADAPTIVELY

A standing challenge in WCOJs is deciding when to use them over binary joins (BJs) as they can make queries slower [17]. This also holds for our factorized aggregation technique, which exploits a delayed expansion over *n:m* joins to calculate (sub) aggregates over a list of join matches once and re-use it. However, if the join turns out to be not *n:m* but *n:1*, these lists have length 1, and this technique only adds overhead. The same holds for our WCOJ algorithm.

Previous work (e.g., [9]) has focused on using the optimizer to create factorized query plans. For this, it should estimate the size of intermediate join results and estimate their cost, or more heuristically, detect patterns such as cycles in join graphs. A big issue for optimizers here is the lack of reliable (or any) statistics to predict join sizes. This is especially hard to do over multiple joins and on inputs computed by sub-queries. Errors made in estimates compound, making them unreliable [12]. This problem is particularly present in modern data pipelines, where DuckDB is often deployed. It is common practice to directly query Parquet files, such that even information about foreign and primary keys is lacking, and it is even hard to identify an *n:m* join in the first place.

We present an adaptive approach that uses runtime statistics to decide whether to employ a factorized query plan. While the query optimizer detects opportunities for factorized execution, it does not commit to factorization immediately. Instead, it marks the plan as *potentially factorizable*, prompting the hash joins to collect statistics during their hash build phase. The optimizer currently targets cyclic joins and changes only the behavior of (i) the join that closes the cycle (the join with a double key condition – the 2nd join in Figure 6), and (ii) the join whose build side delivers the key involved in that lookup (the first join in Figure 6). We refer to this key, labeled as `S.dst` in Figure 6, as the "embedded key." The change introduces the creation of fast *sketches* based on the embedded key during the materialization phase of the joins. We chose the closing of cyclic *n:m* joins as a starting point because it typically reduces intermediate results. Generalizing adaptive factorized processing to different join shapes is left as future work.

For the triangle query in Figure 6, both the binary and factorized join plan begin similarly by building the first hash table on the same key (`S.src = R.dst` from Figure 6), differing only in the probe phase where they either emit chain pointers or expand the result. After building the first join's hash table, there is exact information on the number of chains and their length, plus the information from the sketches on the embedded key (`S.dst`). Next, the second hash table will be built, creating sketches on its respective embedded key (`T.src`). After finishing the materialization of the records and before filling the hash buckets, we can now decide whether to proceed with a BJ or WCOJ plan based on runtime statistics.

Specifically, to change the two normal hash joins into a plan where the first join performs factorized lookups (emits chain pointers) and the second hash join that closes the cycle using a combined key, is turned into another factorized join on only a single key; and runs list intersections to compare with the second key. If this decision is made, mini hash-tables are created on all the chains in both hash tables as a final part of the second hash join's build phase. After this, the probe pipeline will start, looking up each tuple in the first hash table and delivering factorized results containing chain pointers. Subsequently, it will perform a lookup in the second hash table, which delivers another chain pointer, and then it performs a list intersection on both chains to emit flat results.

AMS Sketch. We need to gather statistics regarding the explosiveness of the join pattern that we might handle with a list intersection in the WCOJ plan. For this, we use the AMS sketch, a linear sketch suitable to estimate the join size between relations [2].

The AMS sketch consists of a matrix $M \in \mathbb{Z}^{d \times w}$, where the number of rows d determines the number of hash functions used, and the number of columns w is the number of buckets per hash function. In our implementation, for every item i, we use the 64-bit hash generated during the hash join materialization phase. This hash must be computed in both the binary and the factorized join plan so there is no additional overhead in hashing the keys for the sketch. This 64-bit hash is split into its bytes. For each row j, we consider the j-th byte of this hash. We derive two functions from this byte: (1) The function $h_j(i)$ determines the bucket b in the j-th row to which item i is mapped based on the last 7 bits of the j-th byte. (2) $g_j(i)$ maps item i to either +1 or −1 depending on the most significant bit of the byte. We then increment b with $g_j(i)$.

To estimate the join size of two sketches $M_1, M_2 \in \mathbb{Z}^{d \times w}$, we compute the estimate per row j as $e_j = \sum_{k=1}^{w} M_{1,j,k} \cdot M_{2,j,k}$ using the dot product, and then use the mean of the row estimates.

We construct an AMS sketch on the previously mentioned embedded keys for both join sides during the hash join materialization. For Figure 6, this results in the two sketches $M_{\texttt{S.dst}}$ for the first and $M_{\texttt{T.src}}$ for the second join. The sketches from both hash tables can be used to estimate the result size of the list intersection. Then, by dividing this by the average cardinality of the two build sides, we get the “Chain Intersection Explosion Factor” feature that is less dependent on cardinalities.

We also estimate the self-join size of the chain keys for each join build side by applying its AMS sketch twice. Dividing this by the cardinality of the build side results in the feature “1st Join Chain Keys Skew”.

HLL Sketch. This well-known sketch estimates the number of distinct values. During the build of our Linear-Chained hash table, in the first step of materializing all records, DuckDB already computes the hash on key, so it is cheap to compute an HLL sketch. The number of distinct values can be used to size the bucket array and provide the average chain length.

Along with the statistics from the HLL and AMS sketches, we use simple measures like the average chain length of the first hash table as features for our models, which decide whether to pick a WCOJ or BJ plan.

Feature Overview. Next to the statistics we get from the HLL and the AMS sketches, we use heuristics like the average chain length of the first hash table as features to predict whether to choose a WCOJ or BJ plan. The complete feature list is in Table 1. Additionally, the table presents the Pearson correlation coefficient (PCC) for each feature, illustrating its relationship with the "Faster Class" and the WCOJ over BJ speedup. A positive PCC indicates that the likelihood of WCOJ being faster increases as the feature increases, while a negative correlation suggests the opposite relationship. Our experiments show that computing these sketches and gathering the heuristics only adds an average runtime overhead of 0.35%.

Figure 8 depicts the speedup of triangle queries across different relations compared to the "Both Joins Min Average Chain Length" and the “1st to 2nd Join Build Size Ratio”. The figure shows that the factorized-WCOJ approach does not always outperform the baseline, with speedups ranging from 0.23x to 14.91x. Further, it suggests that runtime parameters correlate with speedup. This aligns with the intuition that longer chains produce large intermediate results, which WCOJ can avoid. Furthermore, looking at the “1st to 2nd Join Build Size Ratio”, we observe that certain features serve as predictors only in specific cases. For instance, this feature is a reliable predictor of when *not* to use WCOJ, namely if its value is smaller than $\simeq 0.05$. The intuition behind this is that the probe side of the 1st join is where WCOJ mitigates the explosion of intermediate results. If the size of the 1st join probe side is too small compared to the 2nd join size, the benefits of WCOJ get smaller. This also shows the potential of combining multiple features in a model.

4 EVALUATION

In this section, we evaluate the performance of our WCOJ implementation for cyclic joins and the benefits of using it adaptively. The cyclic join experiment was run on a cloud instance running Fedora 40, Intel Xeon Gold 5115 with 40 threads and 248 GB RAM. The experiments on adaptivity experiments were performed on a machine running Fedora 40, featuring a 96 threads AMD EPYC CPU and 128 GB RAM.

Cyclic Joins. We evaluated the performance of factorized cyclic joins in DuckDB by performing triangle joins along links in the WebStan dataset[1], which has 685,231 nodes and 7,600,595 edges.

We compared the results with DuckDB v1.0.0 and KùzuDB [13] version 0.4.2.12 commit 89d47d152, an embeddable graph DBMS and Umbra [18] version v0.2-400-g806f09709, a state-of-the-art relational DBMS. Both systems feature a WCOJ implementation. Figure 9 shows the results. In both KùzuDB and DuckDB binary joins were slower than WCOJ. As mentioned previously, KùzuDB required hints to recognize the faster WCOJ plan. DuckDB's factorized WCOJ using list intersection slightly outperformed KùzuDB's WCOJ implementation in both single-threaded and multi-threaded versions. The speedup between the single-threaded and multi-threaded versions of the WCOJ implementation of DuckDB is less than the normal BJ, indicating potential for further optimization, as parts of the algorithm are not executed in parallel.

Umbra's WCOJ performs best with 8 threads, while the binary join is the best in all cases, even outperforming Umbra's WCOJ.

[1] https://snap.stanford.edu/data/web-BerkStan.html

Table 1: Features with Correlations to the Faster Class C_f and the Speedup C_s.

Feature Name	Description	C_f	C_s
1st Join Build Side Cardinality	Size of the build side for the first join.	-0.29	-0.04
1st Join Number of Unique Chains	Number of unique chains in the first join.	-0.31	-0.04
1st Join Average Chain Length	Average Chain Length in the first join.	0.58	0.05
1st Join Chain Keys Skew	Skew of chain keys processed by list intersection, estimated from the 1st AMS sketch self-join size.	-0.02	0.03
2nd Join Build Side Cardinality	Size of the build side for the second join.	-0.24	-0.03
2nd Join Number of Unique Chains	Number of unique chains in the second join.	-0.22	-0.02
2nd Join Average Chain Length	Average chain length in the second join.	-0.08	-0.01
2nd Join Chain Keys Skew	Skew of chain keys processed by list intersection, estimated from the 2nd AMS sketch self-join size.	-0.09	-0.01
Both Joins Average Chain Length	Average chain length across both joins.	0.43	-0.01
Both Joins Average Build Side Cardinality	Average build side cardinality for both joins.	-0.29	-0.04
Larger Join Average Chain Length	Average chain length of the join with the larger build side cardinality.	-0.08	-0.01
Both Joins Max Average Chain Length	Maximum of the two average chain lengths of the two Joins.	-0.08	-0.01
Both Joins Min Average Chain Length	Minimum of the two average chain lengths of the two Joins using an accurate value for the first join and an HLL-based estimation for the second.	0.64	0.01
Both Joins Min Chain Keys Skew	Minimum chain key skew across both joins.	0.11	0.17
Number of Unique Relations	Number of unique relations involved in the triangle query.	-0.34	-0.09
Chain Intersection Join Size	Predicted size the join result processed by list intersection, estimated from the product of the two AMS sketches.	-0.07	-0.01
Chain Intersection Explosion Factor	Chain intersection join size divided by both joins average build side cardinality	0.12	0.17
1st to 2nd Join Build Size Ratio	Ratio of the first join's build side cardinality to the second join's.	0.08	0.09

When running Umbra in the default setting (`set debug.multiway='c'`, which stands for cautious), it chooses a WCOJ plan on this query. However, for Umbra, the WCOJ is the wrong choice; by disabling multiway joins ('d'), the binary join plans appear to be faster, highlighting the optimizer challenges WCOJs pose.

Adaptive Factorization. To evaluate the performance of dynamically choosing between a WCOJ or BJ plan, we compared several heuristics and machine learning models in their ability to predict which plan to use based on statistics gathered during runtime.

We combined three different datasets for this evaluation: (1) A synthetic dataset consisting of 305 graphs, each containing between 100 and 100,000 nodes and between 100 and 10 million edges and a wide range of in- and out-degrees. (2) SNAP datasets [16] featuring more than 1 million edges. (3) LDBC SNB BI [23] datasets of scale factors 1, 3, and 10. The SNB test dataset includes 352 unique queries representing all possible 3-way relational cycles. Each relation is once stored in a DuckDB table and once as a Parquet file[2].

In order to test our approach in various degrees of difficulty, we created four workloads featuring queries on datasets with progressively fewer (useful) static statistics: workload W_1 (n=900) contains triangle queries over PK-FK relations in the SNB datasets, stored as DuckDB base tables, which contain HLL table stats the optimizer leverages for join ordering. W_2 (n=1200) extends W_1 with queries that perform graph joins across tables from the synthetic and SNAP datasets (which are single-edge tables). When joining edges from different datasets using dense integer vertex IDs, joins can still be explosive, but many join keys will not hit anything. We name this workload "Filtering Joins", referring to the sometimes heavy filtering effects experienced in the join - a phenomenon that is hard to predict in advance using static cost models and typically will lead to cardinality estimation errors, a wrong join order, and difficulty in deciding between using WCOJ or BJ based on the statistics available during query optimization. W_3 (n=1800) contains only the SNB (PK-FK) queries from W_1, but now in two variants: on DuckDB base tables (with HLLs) and Parquet files (without HLLs). The effect when joining on Parquet tables is, therefore that there are no table statistics (regarding the number of DISTINCT keys), such that the DuckDB query optimizer creates worse join orders than with base tables. W_4 (n=2400) builds upon W_3 by again adding filtering join queries, both on DuckDB base tables and their Parquet equivalents.

For each query, we measured the runtimes of the BJ and the WCOJ plan to calculate the WCOJ's speedup and identify queries where it outperforms the BJ, serving as target labels for our models. We trained various machine learning models using a 70/30% train-test split, using runtime data and statistics derived from the sketches to construct the features outlined in Table 1.

As our data contains both the WCOJ and BJ runtime for each query, we can additionally simulate scenarios such as the potential speedup when the WCOJ plan is always or never used, which are shown in the top four rows of Table 2.

The best achievable speedup is higher than consistently opting for a WCOJ. For example, in W_1, the best speedup is 1.3x, compared to 1.15x for WCOJ, highlighting the potential of adaptive factorization. We can capitalize on this by using machine learning models to determine the execution strategy on runtime, as shown in Table 2. The random forest model achieves the highest speedup

[2]See https://github.com/cwida/CyclicJoinBench for the full benchmark

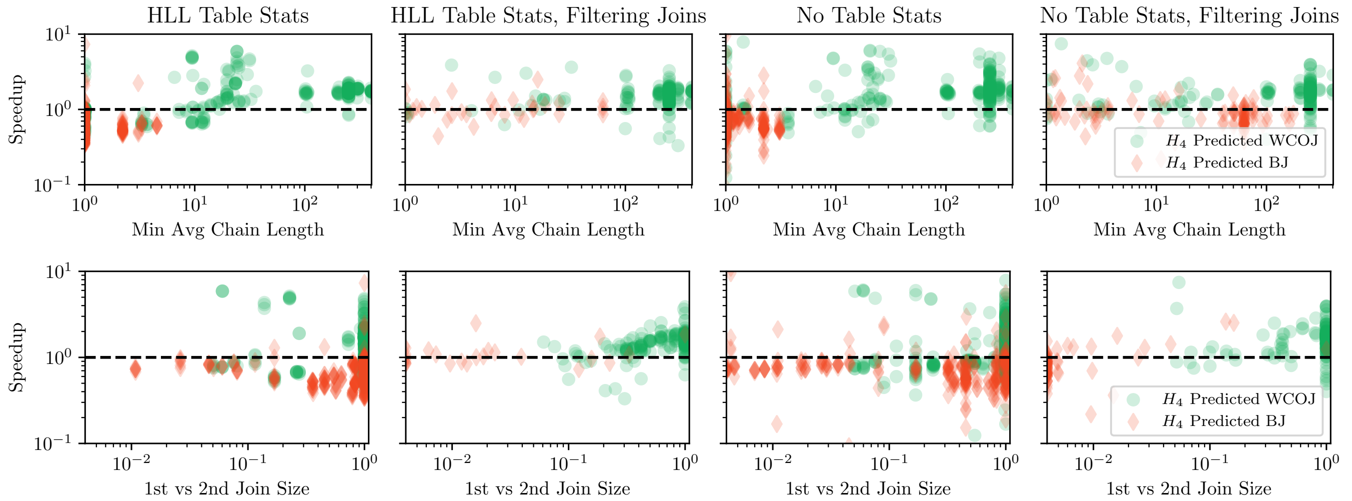

Figure 8: Comparison of WCOJ speedup over BJ with respect to "Both Joins Min Average Chain Length" (top) and "1st to 2nd Join Build Size Ratio" (bottom) for triangle queries on relations from DuckDB tables or Parquet. High correlation is observed with tables due to better join ordering statistics, but it decreases with Parquet files or filters. While minimum chain length roughly correlates with speedup, a low "1st to 2nd Join Build Size Ratio" predicts poor WCOJ performance. Furthermore, we show when H_4 from Table 2 would opt for WCOJ and BJ, showing that we can create good models from these features.

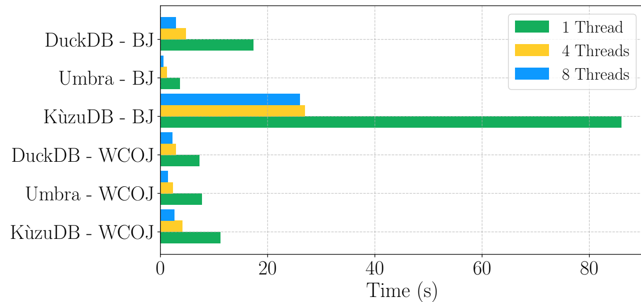

Figure 9: Performance of triangle-count queries in web-BerkStan. DuckDB and KùzuDB profit from WCOJs; Umbra does not. Allocation of mini-hash tables in DuckDB's WCOJ is (still) serial, impairing its parallel scaling.

with 1.29x. The most important features of this model are the "Min Chain Length" and the "1st Join Chain Length" as depicted in Figure 10. However, with 98 nodes, this model is already quite complex and counter-intuitive and bears the risk of over-fitting (despite our 70/30 train/test approach) and hinders system adoption. In contrast, the decision tree shown in Figure 11 might be a better choice as it offers similar performance with only 9 nodes. Like the random forest, one of the key features used is the "Min Chain Length", opting for a WCOJ if chains are longer than 5.

Also, simpler heuristics (e.g., use WCOJ if a feature exceeds a threshold) can achieve good speedups. For instance, H_1 using only the first join's average chain length reaches 1.27x speedup on DuckDB tables (W_1). However, it performs worse on Parquet files (W_3). When running queries on tables, the optimizer can use HLL table statistics to delay explosive joins. Therefore, if the first join has longish chains, subsequent joins likely do as well, making the first join's chain length a strong predictor. This heuristic does not work with poor join ordering. Further, we can improve the decision-making by combining multiple features, as in H_4, yielding good performance in all datasets, as shown by the colors in Figure 8. Importantly, the "MinChainLength" feature needs an estimate of the chain length of the second join, which can be provided by *run-time* computation of an HLL during hash-build. Similarly "MinKeySkew" needs an AMS sketch on both join inputs.

Table 2: Speedup of WCOJ over BJ for different models for adaptive decision-making across different workloads W_1 to W_4 with progressively less static statistics for join ordering. With good join ordering, simple heuristics like H_1 perform well, while bad join ordering requires more complex models.

Selection Method	W_1	W_2	W_3	W_4
Static Models Speedup				
Always use slower algorithm	0.84	0.86	0.83	0.87
Always use faster algorithm	1.30	1.42	1.39	1.47
Always use WCOJ	1.15	1.28	1.23	1.34
Always use BJ	1.00	1.00	1.00	1.00
Heuristics Speedup, Use WCOJ if True				
H_1 = `1stJoinChainLength` > 10	1.27	1.36	1.29	1.37
H_2 = `MinChainLength` > 5.5	1.27	1.37	1.29	1.37
$H_3 = H_2 \vee$ `MinKeySkew` > 5	1.26	1.36	1.32	1.40
$H_4 = H_3 \wedge$ `JoinSizeRatio` > 0.05	**1.27**	**1.37**	**1.35**	**1.42**
Machine Learning Models Speedup				
Decision Tree	1.26	1.38	**1.34**	**1.41**
Logistic Regression	1.22	1.30	1.27	1.31
Random Forest	**1.29**	**1.39**	1.30	1.37
Gradient Boosting	1.29	1.39	1.29	1.38

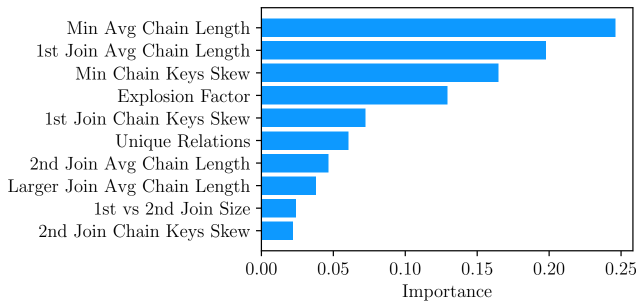

Figure 10: The feature importance of the random forest shows that the "Minimum Average Chain Length" and the "1st Join Average Chain Length" are the most relevant.

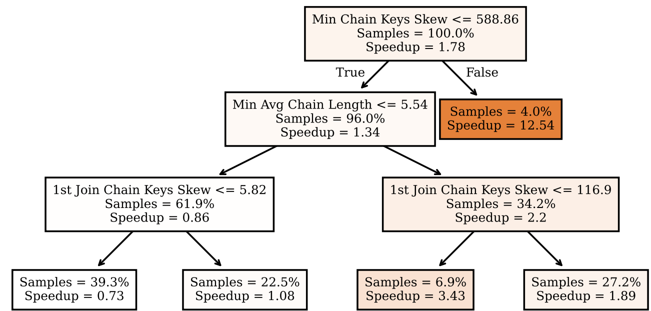

Figure 11: The decision tree identifies the "Minimum Average Chain Length" and "Chain Key Skew" as important features for decision-making.

We think heuristics (or simple models) are preferred over learned models in real systems, as they are understandable by developers. In this regard, it is a happy coincidence that H_4 happens to outperform our ML models. We would still call for caution regarding the generality of H_4's performance, which remains to be battle-tested in real deployment, but we think it provides a good starting point.

5 CONCLUSION

We presented the Linear-Chained hash table design, which was merged into DuckDB v1.1.0 without performance regressions over the previous bucket-chained design. Its collision-free chains are a practical form of so-called *d-representations* in factorized query processing. We used them to implement a proof-of-concept WCOJ and factorized aggregation that is closely integrated with the existing binary hash join in DuckDB, with low impact on its source code.

Identifying opportunities where WCOJ and factorization accelerate queries are beneficial is an unsolved optimizer challenge. This paper shows that runtime adaptivity can help: in the build phase, one can inspect all build-side tuples and construct AMS and HLL sketches. These statistics are created on the *true* join inputs; whereas statistics on base tables – often unavailable in the first place – get perturbed by operators in between the scan and the join, causing estimation errors. Using the statistics of our sketches, we were able to create a simple and robust model to predict whether WCOJs on cyclic joins improve DuckDB performance.

Future Work. We are still working to improve the Linear-Chained hash table on the implementation level, and also plan to test an "unchained" [4] variant that clusters all records for a duplicated key together to improve memory locality. However, unlike [4], to avoid regressions, we think this should only be done *adaptively*, e.g., if an HLL sketch sees evidence of long chains. This is easier to integrate in vectorized engines like DuckDB than in compiling engines like Umbra: to use runtime adaptivity, the latter must perform JIT recompilation after making runtime decisions or include the code for both branches, causing code growth exponential in the number of decisions. Both issues may increase compilation latency.

Our next step is integrating factorized techniques into DuckDB or DuckPGQ, enabling factorized aggregation to adapt dynamically to runtime statistics and compose with non-factorized components. We will also explore supporting more complex join shapes by enhancing our hash table-based d-representation.

REFERENCES

[1] Daniel Abadi et al. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* (2013), 197–280.
[2] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In *STOC*. ACM, 20–29.
[3] Nurzhan Bakibayev et al. 2012. FDB: A Query Engine for Factorised Relational Databases. *Proc. VLDB Endow.* (2012).
[4] Altan Birler et al. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *DaMoN*. ACM, 4:1–4:9.
[5] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proceedings of the VLDB Endowment* 17, 1 (2024).
[6] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. 225–237.
[7] Jemery Chen et al. 2023. Accurate Summary-based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *SIGMOD Rec.* (2023), 94–102.
[8] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D Hash Join: Building On Non-Unique Join Attributes. In *CIDR*.
[9] Michael J. Freitag et al. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* (2020), 1891–1904.
[10] Michael J. Freitag and Thomas Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *CIDR*.
[11] Pranjal Gupta et al. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504. https://doi.org/10.14778/3476249.3476297
[12] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*. ACM Press, 268–277.
[13] Guodong Jin et al. 2023. KÙZU Graph Database Management System. In *CIDR*.
[14] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *ICDE*. IEEE.
[15] 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 *SIGMOD*.
[16] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[17] Amine Mhedhbi. 2023. *GraphflowDB: Scalable Query Processing on Graph-Structured Relations*. Ph. D. Dissertation. University of Waterloo.
[18] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[19] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* (2018), 16:1–16:40.
[20] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* (2016), 5–16.
[21] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*.
[22] Siddhartha Sahu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* (2020), 595–618.
[23] Gábor Szárnyas et al. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* (2022), 877–890.
[24] Daniel ten Wolde, Gábor Szárnyas, and Peter A. Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *Proc. VLDB Endow.* (2023), 4034–4037.
[25] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106.