Free Join: Unifying Worst-Case Optimal and Traditional Joins

YISU REMY WANG, University of Washington, USA
MAX WILLSEY, University of Washington, USA
DAN SUCIU, University of Washington, USA

Over the last decade, worst-case optimal join (WCOJ) algorithms have emerged as a new paradigm for one of the most fundamental challenges in query processing: computing joins efficiently. Such an algorithm can be asymptotically faster than traditional binary joins, all the while remaining simple to understand and implement. However, they have been found to be less efficient than the old paradigm, traditional binary join plans, on the typical acyclic queries found in practice. Some database systems that support WCOJ use a hybrid approach: use WCOJ to process the cyclic subparts of the query (if any), and rely on traditional binary joins otherwise. In this paper we propose a new framework, called Free Join, that unifies the two paradigms. We describe a new type of plan, a new data structure (which unifies the hash tables and tries used by the two paradigms), and a suite of optimization techniques. Our system, implemented in Rust, matches or outperforms both traditional binary joins and WCOJ on standard query benchmarks.

CCS Concepts: • **Information systems** → **Join algorithms**.

Additional Key Words and Phrases: Worst-case optimal join

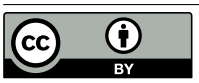
ACM Reference Format:
Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2, Article 150 (June 2023), 23 pages. https://doi.org/10.1145/3589295

1 INTRODUCTION

Over the last decade, worst-case optimal join (WCOJ) algorithms [19–21, 27] have emerged as a breakthrough in one of the most fundamental challenges in query processing: computing joins efficiently. Such an algorithm can be asymptotically faster than traditional binary joins, all the while remaining simple to understand and implement [21]. These algorithms opened up a flourishing field of research, leading to both theoretical results [14, 21] and practical implementations [2, 7, 17, 27].

Over time, a common belief took hold: "WCOJ is designed for cyclic queries". This belief is rooted in the observation that WCOJ enjoys lower asymptotic complexity than traditional algorithms for cyclic queries [21], but when the query is acyclic, classic algorithms like the Yannakakis algorithm [28] are already asymptotically optimal. Moreover, traditional binary join algorithms have benefited from decades of research and engineering. Techniques like column-oriented layout, vectorization, and query optimization have contributed compounding constant-factor speedups, making it challenging for WCOJ to be competitive in practice. This has lead many instantiations of WCOJ, including Umbra [7], Emptyheaded [2], and Graphflow [17], to adopt a hybrid approach: using WCOJ to process parts of the query, and resorting to traditional algorithms (usually binary join) for the rest. Having two different algorithms in the same system requires changing and

Authors' addresses: Yisu Remy Wang, University of Washington, USA, remywang@cs.washington.edu; Max Willsey, University of Washington, USA, mwillsey@cs.washington.edu; Dan Suciu, University of Washington, USA, suciu@cs.washington.edu.



2836-6573/2023/6-ART150
https://doi.org/10.1145/3589295

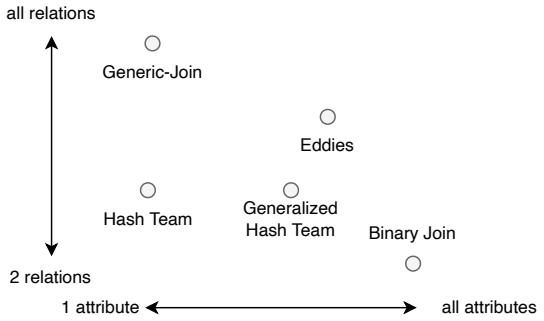


Fig. 1. Design space of join algorithms.

potentially duplicating existing infrastructure like the query optimizer. This introduces complexity, and hinders the adoption of WCOJ.

The dichotomy of WCOJ versus binary join has led researchers and practitioners to view the algorithms as opposites. In this paper, we break down this dichotomy by proposing a new framework called Free Join that unifies WCOJ and binary join. We propose several new techniques to make Free Join outperform both binary join and WCOJ: we design an algorithm to convert any binary join plan to a Free Join plan that runs as fast or faster; we design a new data structure called COLT (for *Column-Oriented Lazy Trie*), adapting the classic column-oriented layout to improve the trie data structure used in WCOJ; and we propose a vectorized execution algorithm for Free Join.

To explain these contributions we provide some context. In this paper, we focus on algorithms based on hashing, and choose Generic Join [21] as a representative of WCOJ algorithms. A crucial difference between Generic Join and binary join lies in the way they process each join operation. Binary join processes two relations at a time, and joins on *all attributes* in the join condition between these two relations. In contrast, Generic Join processes one attribute at a time, and joins *all relations* that share that attribute. This suggests a design space of join algorithms, where each join operation may process any number of attributes and relations. Figure 1 shows this design space which also covers classic multiway join algorithms like Hash Team [9], Generalized Hash Team [12] and Eddies [4]. Being able to join on any number of variables and relations frees us from the constraints of all existing algorithms mentioned above.

Our new framework, Free Join, covers the entire design space, thereby generalizing and unifying existing algorithms. The starting observation is that the execution of a left-deep linear binary join plan is already very similar to Generic Join. While Generic Join (reviewed in Sec. 2) is traditionally specified as a series of nested loops [21], the push-based model [13, 18] for executing a left-deep linear binary plan is also implemented, similarly, as nested loops. The two algorithms also process each join operation similarly: each binary hash join iterates over tuples on one relation, and for each tuple probes into the hash table of another relation; each loop level in Generic Join iterates over the keys of a certain trie, and probes into several other tries for each key. This inspired us to unify hash tables and hash tries into the same data structure, and develop Free Join using iteration and probing as key operations. This finer-grained view of join algorithms allows Free Join to generalize and unify existing algorithms, while precisely capturing each of them.

Free Join takes as input an already optimized binary join plan, and converts it into a new kind of plan that we call a Free Join plan. It then optimizes the Free Join plan, resulting in a plan that sits in between binary join and Generic Join, combining the benefits of both. On one hand Free Join takes full advantage of the design space in Figure 1. On the other hand, by starting from an already

optimized binary plan, Free Join takes advantage of existing cost-based optimizers; in our system we used binary plans produced by the optimizer of DuckDB [24, 25].

Next, we address the main source of inefficiency in Generic Join: the need to construct a trie on each relation in the query. In contrast, a binary join plan needs to build a hash map only for each relation on the right-hand side of a join, and simply iterates over the relation on the left. In practice, trie-building has been observed to be a major bottleneck for Generic Join [7, 17], making it slower than binary join. This is because each trie is more expensive to build than a hash map, and the left relation is usually chosen to be a large relation by the query optimizer. One simple optimization in Free Join is that we do not built a trie for tables that are left children, mimicking the binary plans. However, we go a step further, and introduce the *Column-Oriented Lazy Trie* (COLT) data structure, which builds the inner subtries lazily, by creating each subtrie on demand. We note that this builds on an earlier idea in [7]. As the name suggests, COLT adapts the lazy trie data structure in [7] to use a column-oriented layout. And unlike the original lazy trie which builds at least one trie level per table, COLT completely eliminates the cost of trie building for left tables.

Finally, we describe a method for incorporating vectorized processing in Free Join, allowing it to collect multiple data values before entering the next iteration level. The standard Generic Join processes one data value at a time, but, as is the case in traditional query engines, this leads to poor cache locality. Vectorized execution [23] was proposed for binary join to improve its locality by processing data values in batch. By breaking down join operations into iterations and probes, Free Join gives rise to a simple vectorized execution algorithm that breaks each iteration into chunks and groups together batches of probes. Our proposal is to our knowledge the first vectorized execution algorithm for Generic Join.

We implemented Free Join as a standalone Rust library, and compared it with two baselines: (1) our own Generic Join implementation in Rust, and (2) the binary hash join implemented in DuckDB [24, 25], a state-of-the-art in-memory database. We found that, on acyclic queries, Free Join is up to 19.36x faster than binary join, and up to 31.6x faster than Generic Join; on cyclic queries, Free Join is up to 15.45x faster than binary join, and up to 4.08x faster than Generic Join.

While optimizers for binary plans have been developed and improved over decades [26], little is known about optimizing Generic Join. A Generic Join plan consists of a total order on its variables, and its run time does depend on the choice of this order. But since the theoretical analysis of Generic Join guarantees worst case optimality for *any* variable order, it is a folklore belief that Generic Join is more robust than binary join plans to poor choices of the optimizer. We also conduced experiments measuring the robustness of the three types of plans (binary, Generic Join, Free Join) to poor choices of the optimizer. We found that Generic Join is indeed the least sensitive, while Free Join, like binary joins, suffers more from the poor optimization choices of the optimizer, since both rely on a cost-based optimized plan. However, Generic Join starts from worse baseline than Free Join. In other words, Free Join takes better advantage of a good plan, when available, than Generic Join does.

In summary, we make the following contributions in this paper:

(1) Free Join, a framework unifying existing join algorithms (Section 3).
(2) An algorithm to converting any binary join plan into an optimized Free Join plan (Section 4.1).
(3) COLT, a column-oriented lazy trie data structure (Section 4.2).
(4) A vectorized execution algorithm for Free Join (Section 4.3).
(5) Experiments evaluating the algorithms and optimizations (Section 5).

2 BACKGROUND

This section defines basic concepts and reviews background on binary join and Generic Join.

2.1 Basic Concepts

We consider a relational database where each relation has a fixed *schema*, and may have duplicates, i.e. we use bag semantics. A *full conjunctive query* has the following form:

$$Q(\boldsymbol{x}) \text{ :- } R_1(\boldsymbol{x}_1), \ldots, R_m(\boldsymbol{x}_m). \tag{1}$$

Each term $R_i(\boldsymbol{x}_i)$ is called an *atom*, where R_i is a relation name and $\boldsymbol{x}_i$ a tuple of variables. The query is *full*, meaning that the head variables $\boldsymbol{x}$ include all variables appearing in the atoms. To reduce clutter in the following sections, we will assume that the query does not have self-joins. This is without loss of generality: if two atoms have the same relation name, then we simply rename one of them. Our system also supports selections, projections, and aggregation. We assume that the selections are pushed down to the base tables, thus the atom R_i in (1) may include a selection over a base table; in particular, all variables in the atom $R_i(\boldsymbol{x}_i)$ are distinct. Similarly, projections and aggregates are performed after the full join, hence none of them is shown in (1).

Example 2.1. Consider the following SQL query:

```
SELECT r.x, s.u, t.u
  FROM R as r, M as s, M as t -- schema: R(x,y), M(u,v,w)
 WHERE s.w > 30 AND t.v = t.w
   AND r.y = s.u AND s.v = t.u AND t.v = r.x
```

Then we denote by $S = \Pi_{uv}(\sigma_{w>30}(M))$ and $T = \Pi_{uv}(\sigma_{v=w}(M))$, and write the query as:

$$Q_\triangle(x, y, z) \text{ :- } R(x, y), S(y, z), T(z, x).$$

We call this query the *triangle query* over the relations R, S, T.

It is often convenient to view the conjunctive query (1) as a hypergraph. The *query hypergraph* of Q consists of vertices $\mathcal{V}$ and edges $\mathcal{E}$, where the set of nodes $\mathcal{V}$ is the set of variables occurring in Q, and the set of hyperedges $\mathcal{E}$ is the set of atoms in Q. The hyperedge associated to the atom $R(\boldsymbol{x}_i)$ is defined as the set consisting of the nodes associated to the variables $\boldsymbol{x}_i$. As standard, we say that the query Q is *acyclic* if its associated hypergraph is α-acyclic [5]

2.2 Binary Join

The standard approach to computing a conjunctive query (1) is to compute one binary join at a time. A *binary plan* is a binary tree, where each internal node is a join operator $\bowtie$, and each leaf node is one of the base tables (atoms) $R_i(\boldsymbol{x}_i)$ in the query (1). The plan is a *left-deep linear plan*, or simply left-deep plan, if the right child of every join is a leaf node. If the plan is not left-deep, then we call it *bushy*. For example, $(R \bowtie S) \bowtie (T \bowtie U)$ is a bushy plan, while $((R \bowtie S) \bowtie T) \bowtie U$ is a left-deep plan. We do not treat specially right-deep or zig-zag plans, but simply consider them to be bushy.

In this paper we consider only hash-joins, which are the most common types of joins in database systems. The standard way to execute a bushy plan is to decompose it into a series of left-deep linear plans. Every join node that is a right child becomes the root of a new subplan, which is first evaluated, and its result materialized, before the parent join can proceed. As a consequence, every binary plan, bushy or not, becomes a collection of left-deep plans. We decompose bushy plans in exactly the same way, and we will focus on left-deep linear plans in the rest of this paper. For example, the bushy plan $(R \bowtie S) \bowtie (T \bowtie U)$ is converted into two plans: $P_1 = T \bowtie U$ and $P_2 = (R \bowtie S) \bowtie P_1$; both are left-deep plans.

To reduce clutter, we represent a left-deep plan $(\cdots((R_1 \bowtie R_2) \bowtie R_3) \cdots \bowtie R_{m-1}) \bowtie R_m$ as $[R_1, R_2, \ldots, R_m]$. Evaluation of a left-deep plan is done using pipelining. The engine iterates over

```
for (x, y) in R:
  s = S[y]?
  for (y, z) in s:
    t = T[x,z]?
    for (x, z) in t:
      output(x, y, z)
```

(a) Binary join.

```
for a in R.x ∩ T.x:
  r = R[a]; t = T[a]
  for b in r.y ∩ S.y:
    s = S[b]
    for c in s.z ∩ t.z:
      output(a, b, c)
```

(b) Generic Join.

Fig. 2. Execution of binary join and Generic Join for $Q_\triangle$. The notation `S[y]?` performs a lookup on S with the key y, and continues to the enclosing loop if the lookup fails. Binary join iterates over *tuples*, Generic Join iterates over *values*.

each tuple in the left-most base table R_1; each tuple is probed in R_2; each of the matching tuple is further probed in R_3, etc.

Example 2.2. A possible left-deep linear plan for $Q_\triangle$ is $[R, S, T]$, which represents $(R(x, y) \bowtie S(y, z)) \bowtie T(z, x)$. To execute this plan, we first build a hash table for S keyed on y, where each y maps to a vector of (y, z) tuples, and a hash table for T keyed on x and z, each mapped to a vector of (x, z) tuples[1]. Then the execution proceeds as shown in Figure 2a. For each tuple (x, y) in R, we first probe into the hash table for S using y to get a vector of (y, z) tuples. We then loop over each (y, z) and probe into the hash table for T using x and z. Each successful probe will return a vector of (x, z) tuples, and we output the tuple (x, y, z) for each (x, z).

2.3 Generic Join

Generic Join was introduced in [21] and is the simplest worst-case optimal join algorithm. It is based on the earlier Leapfrog Triejoin algorithm [27]. Generic Join computes the query Q in (1) through a series of nested loops, where each loop iterates over a variable (not a tuple). Concretely, Generic Join chooses arbitrarily a variable x, computes the intersection of all x-columns of all relations containing x, and for each value a in this intersection it computes the residual query $Q[a/x]$, where every relation R that contains x is replaced with $\sigma_{x=a}(R)$. In pseudocode:

```
GJ: for a in ⋂{Π_x(R_i) | R_i contains x}
      compute Q[a/x]  \\ run GJ on Q with one fewer variable
```

If the query Q has k variables, then there are k nested loops in Generic Join. In the inner most loop, Generic Join outputs the tuple of constants, one from each iteration.[2] We notice that a plan for Generic Join consists of a total order of the variables of the query, which we denote as $[x_1, x_2, \ldots, x_k]$. Assuming that the intersection above is done optimally (see below), the algorithm is provably worst-case-optimal, for any choice of the variable order.

Example 2.3. Fig. 2b illustrates the pseudocode for Generic Join on the query $Q_\triangle$, using the variable order $[x, y, z]$. We denoted $\Pi_x(R)$ by $R.x$, and denoted (with some abuse) $\sigma_{x=a}(R)$ by $R[a]$.

While binary joins use hash tables, an implementation of Generic Join uses a *hash trie*, one for each relation in the query. The hash-trie is a tree, whose depth is equal to one plus the number

[1]When the relations are bags, then the hash table may contain duplicate tuples, or store separately the multiplicity. We also note that the question what exactly to store in the hash table (e.g. copies of the tuples, or pointers to the tuple in the buffer pool) has been studied for a long time, see [8].
[2]For bag semantics, it multiplies their multiplicities.

of attributes of the relation, and where each node is either an empty leaf node,[3] or a hash map mapping each atomic value to another node. We will call the *level* of a node to be the distance from the root, i.e. the root has level 0, its children level 1, etc. The hash-trie completely represents the relation: every root-to-leaf path corresponds precisely to one tuple in the relation. Generic Join uses the hash-trie as follows. In order to compute $\sigma_{x=a}(R)$, it simply probes the current hash table for the value $x = a$, and returns the corresponding child. To compute an intersection $\Pi_x(R_1) \cap \Pi_x(R_2) \cap \cdots$, it selects the trie with the fewest keys, say R_1, then iterates over every value a in the keys for R_1 and probes it in each of the hash-maps for $R_2, R_3, \ldots$; this is a provably optimal algorithm for the intersection.

Example 2.4. Consider the query $Q_\triangle$ and the Generic Join plan $[x, y, z]$. We first build a hash trie each for R, S, and T. Each trie has three levels including the leaf. Level 0 of R is keyed on x, level 1 is keyed on y, level 2 contains empty leaf nodes, and similarly for S and T. Consider again the pseudocode in Figure 2b. The first loop intersects level 0 of the R-trie and the T-trie. For each value a in the intersection, we retrieve the corresponding children $R[a]$ and $T[a]$ respectively; these are at level 1. The second loop intersects the hash map $R[a]$ (at level 1) with the level 0 hash-map of S. For each value b in the intersection it retrieves the corresponding children (levels 2 and 1 respectively), and, finally, the innermost loop intersects the S- and T-hash maps (both at level 2), and outputs (a, b, c) for each c in the intersection. So far we have assumed set semantics; if the relations have bag semantics, then we simply multiply the tuple multiplicities on the leaves (level 3).

2.4 Binary Join v.s. Generic Join

Binary join and Generic Join each have their own advantages and disadvantages. Generic Join became popular because of its asymptotic performance guarantee: Ngo et al. [21] proved the algorithm is *worst-case optimal* for *any variable order*, in the sense that its run time is bounded by the largest possible size of its output, called AGM bound [3]. For example, Generic Join executes $Q_\triangle$ in time $\sqrt{|R| \cdot |S| \cdot |T|}$, which is $n^{3/2}$ when all relations have size n; in contrast, a binary join plan can take $\Omega(n^2)$. We note, however, that this formula does not include the preprocessing time needed to construct the tries. For example, if T is significantly larger than R, S, then the run time of Generic Join is $\ll |T|$, yet during preprocessing Generic Join needs to read the entire relation T. On the other hand, binary join has been a staple of database systems for decades. The hash table data structure is simpler than hash tries and is cheaper to build. Techniques like vectorized execution and column-oriented layout have also made binary join practically efficient, but these optimizations have not been adapted for Generic Join. Binary join plans are known to be very sensitive to the choice of the optimizer: poor plans perform catastrophically bad [15]. In contrast, although the runtime performance of Generic Join does depend on the variable order, some researchers believe that Generic Join is less sensitive to poor variable orders, in part because it is always theoretically optimal.

3 FREE JOIN

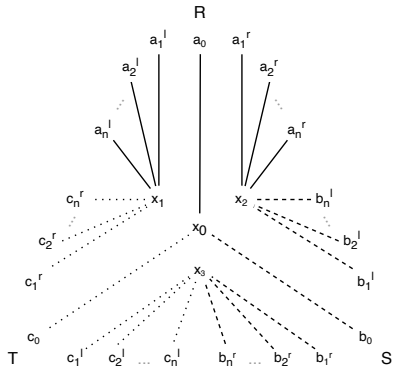
In this section we introduce the Free Join framework. We start by presenting the Generalized Hash Trie (GHT) which is the data structure used in Free Join (Section 3.1). Next we introduce the Free Join plan that specifies how to execute a query with Free Join (Section 3.2). Finally, we describe the Free Join algorithm, which takes as input a collection of GHTs and a Free Join plan, and computes the query according to the plan (Section 3.3).

[3]For bag semantics, we store in the leaf the multiplicity of the tuple.

$$Q_\clubsuit(x,a,b,c) :- R(x,a), S(x,b), T(x,c)$$

$$R = \{(x_0, a_0)\} \cup \{(x_1, a_i^l), (x_2, a_i^r) \mid i \in [1 \ldots n]\}$$
$$S = \{(x_0, b_0)\} \cup \{(x_2, b_i^l), (x_3, b_i^r) \mid i \in [1 \ldots n]\}$$
$$T = \{(x_0, c_0)\} \cup \{(x_3, c_i^l), (x_1, c_i^r) \mid i \in [1 \ldots n]\}$$

(a) $Q_\clubsuit$ and inputs.



(b) Visualization of the input relations.

Fig. 3. (3a) the clover query $Q_\clubsuit$, and an input instance. (3b) visualization of the instance in Fig. 3a. The solid (top) edges form the relation R, the dashed (right) edges form the relation S, and the dotted (left) edges form the relation T. The relations join on the attribute in the center. The only output tuple consists of the three edges in the center.

```
interface GHT {
  # fields
  relation: String, vars: Vec<String>
  # constructor
  fn new(name: String, schema: Vec<Vec<String>>) -> Self
  # methods
  fn iter() -> Iterator<Tuple>
  fn get(key: Tuple) -> Option<GHT> }
```

Fig. 4. The GHT interface.

We will show how each of the above components generalizes and unifies the corresponding components in binary join and Generic Join: the GHT generalizes hash tables and hash tries, the Free Join plan generalizes binary plans and Generic Join plans, and the Free Join algorithm generalizes binary join and Generic Join.

Throughout this section we will make use of the *clover query* $Q_\clubsuit$ in Figure 3a. Figure 3b visualizes the input relations for this query. Note that $Q_\clubsuit$ is *acyclic*.

3.1 The Generalized Hash Trie

To unify binary join and Generic Join, we first need to unify the data structures they work over. We propose the Generalized Hash Trie which generalizes both the hash table used in binary join and the hash trie used in Generic Join.

Definition 3.1 (Generalized Hash Trie (GHT)). A GHT is a tree where each leaf is a vector of tuples, and each internal node is a hash map whose keys are tuples, and each key maps to a child node.

We will reuse the terminology defined for tries, including *level*, *node*, and *leaf*, etc., for GHTs. We will also use the terms GHT and *trie* interchangeably when the context is clear. The *schema* of a GHT is the list $[\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_\ell]$ where $\boldsymbol{y}_k$ are the attribute names of the key at level k.

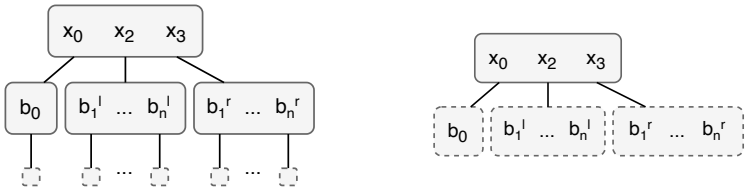


Fig. 5. Two GHTs. The one on the left is also a hash trie, and the one on the right is similar to a hash table. Each box with solid border stores hash keys, and each box with dashed border is a vector of tuples. An empty box is an empty vector, representing a leaf.

The hash trie used in Generic Join is a GHT where each key is a tuple of size one, and the last level stores empty vectors, each of which represents a leaf. The hash table used in binary join is very similar to a GHT with only two levels, where level 0 stores the keys and level 1 stores vectors of tuples. A small difference is that, in the GHT, the concatenation of a tuple from level 0 with a tuple from level 1 forms a tuple in the relation, whereas each whole tuple is stored directly in a hash table. We will show in Section 4.2 how the COLT data structure more faithfully captures the structure of a hash table. Figure 5 shows two examples of GHTs.

We use GHTs to represent relations, and attach metadata as well as access methods to each GHT, to be used by the Free Join algorithm. The GHT interface is shown in Figure 4. The `relation` field stores the relation name. A sub-trie inherits its name from its parent. The `vars` field stores parts of the relation's schema: if the trie is a vector of tuples, `vars` is the schema of each tuple; if the trie is a map, `vars` is the schema of each key. The constructor method `new` creates a new GHT from the named relation, where an n-th level trie has variables matching the n-th element of the `schema` argument, and the values along each path from the root to a leaf of the GHT form a tuple in the relation.

Example 3.2. Both GHTs in Figure 5 represent relation S from the clover query $Q_\clubsuit$ in Figure 3a. The GHT on the left (a hash trie) was created by calling the constructor method `new` with the schema `[[x],[b],[]]`, so the top-level trie has the schema `[x]`, each second-level trie has the schema `[b]`, and each third-level trie (a leaf) has the empty schema `[]`. The GHT on the right (a hash table) was created by calling `new` with the schema `[[x],[b]]`. It has only two levels, with schema `[x]` and `[b]`, respectively. Note that each b value in the hash trie is hashed and stored as a key, while the b values in the hash table are simply stored in vectors.

The methods `iter` and `get` provide access to values stored in the trie. If the trie is a map, `get(key)` returns the sub-trie mapped to `key`, if any. Calling `get` on a vector returns `None`. If the trie is a vector, `iter()` returns an iterator over the tuples in the vector; calling `iter` on a map returns an iterator over the keys.

Example 3.3. On the second GHT in Figure 5, calling `iter` returns an iterator over the values $[x_0, x_2, x_3]$. Calling `get` with the key x_2 returns the sub-trie which is the vector $[b_1^l, \ldots, b_n^l]$. Calling `iter` on this sub-trie returns an iterator over $[b_1^l, \ldots, b_n^l]$.

3.2 The Free Join plan

A Free Join plan specifies how the Free Join algorithm should be executed. It generalizes and unifies binary join plans and Generic Join plans. Recall that a left-deep linear plan for binary join is a sequence of relations; it need not specify the join attributes, since all shared attributes are joined. In contrast, a Generic Join plan is a sequence of variables; it need not specify the relations, since all

relations on each variable are joined. A Free Join plan may join on any number of variables and relations at each step, and therefore needs to specify both explicitly.

To help define the Free Join plan, we introduce two new concepts, called *subatom* and *partitioning*. Fix the query Q in Eq. (1):

Definition 3.4. A *subatom* of an atom $R_i(\boldsymbol{x}_i)$ is an expression $R_i(\boldsymbol{y})$ where $\boldsymbol{y}$ is a subset of the variables $\boldsymbol{x}_i$. A *partitioning* of the atom $R_i(\boldsymbol{x}_i)$ is a set of subatoms $R_i(\boldsymbol{y}_1), R_i(\boldsymbol{y}_2), \ldots$ such that $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ are a partition of $\boldsymbol{x}_i$.

We now define the Free Join plan using these concepts.

Definition 3.5 (Free Join Plan). Fix a conjunctive query Q. A Free Join *plan* is a list $[\phi_1, \ldots, \phi_m]$, where each ϕ_k is a list of subatoms of Q, called a *node*. The nodes are required to *partition the query*, in the sense that, for every atom $R_i(\boldsymbol{x}_i)$ in the query, the set of all its subatoms occurring in all nodes must form a partitioning of $R_i(\boldsymbol{x}_i)$. We denote by $vs(\phi_k)$ the set of variables in all subatoms of ϕ_k. The variables *available to* ϕ_k are all variables of the preceding nodes:

$$avs(\phi_k) = \bigcup_{j<k} vs(\phi_j)$$

We will define shortly a *valid plan*, but first we show an example.

Example 3.6. The following is an Free Join plan for $Q_\clubsuit$:

$$[[R(x,a), S(x)], [S(b), T(x)], [T(c)]] \tag{2}$$

To execute the first node we iterate over each tuple (x, a) in R and use x to probe into S; for each successful probe we execute the second node: we iterate over each b in $S[x]$, then use x to probe into T; finally the third node iterates over c in $T[x]$. The reader may notice that this corresponds precisely to the left-deep plan $(R(x,a) \bowtie S(x,b)) \bowtie T(x,c)$. Another Free Join plan for $Q_\clubsuit$ is:

$$[[R(x), S(x), T(x)], [R(a)], [S(b)], [T(c)]] \tag{3}$$

This plan corresponds to the Generic Join plan $[x, a, b, c]$. Intuitively, here we start by intersecting $R.x \cap S.x \cap T.x$, then, for each x in the intersection, we retrieve the values of a, b, and c from R, S, and T, and output their Cartesian product.

Not all Free Join plans are valid, and only valid plans can be executed. We execute each Free Join node by iterating over one relation in that node, and probe into the others. Therefore, the values used in each probe must be available, either from the same node or a previous one.

Definition 3.7. A Free Join plan is *valid* if for every node ϕ_k the following two properties hold. (a) No two subatoms share the same relation, and (b) there is a subatom containing all variables in $vs(\phi_k) - avs(\phi_k)$. We call such an subatom a *cover* for ϕ_k, and write $cover(\phi_k)$ for the set of covers.

We will assume only valid plans in the rest of the paper. To simplify the presentation, in this section we assume that each node Φ_k, has *one* subatom designated as cover, and will always list it as the first subatom in Φ_k. We will revisit this assumption in Sec. 4, and allow for multiple covers.

Example 3.8. Both plans in Example 3.6 are valid. The covers for the 3 nodes for Eq. (2) are $R(x,a)$, $S(b)$, and $T(c)$, respectively. For the plan in Eq. (3), the covers for the 4 nodes are $R(x), R(a), S(b), T(c)$; for the first node we could have also chosen $S(x)$ or $T(x)$ as cover.

Example 3.9. An example of an *invalid* plan for the clover query has one single node containing all relations and variables:

$$[[R(x,a), S(x,b), T(x,c)]]$$

Intuitively, we cannot execute it: if we iterate over, say R, then we bind two variables x and a, but to lookup S we need the key (x, b).

```
1  fn join(all_tries, plan, tuple):
2    if plan == []:
3      output(tuple)
4    else:
5      tries = [ t ∈ all_tries | t.relation ∈ plan[0] ]
6      # iterate over the cover
7      @outer for t in tries[0].iter():
8        subtries = [ iter_r.get(t) ]
9        tup = tuple + t
10       # probe into other tries
11       for trie in tries[1..]:
12         key = tup[trie.vars]
13         subtrie = trie.get(key)
14         if subtrie == None: continue @outer
15         subtries.push(subtrie)
16       new_tries = all_tries[tries ↦ subtries]
17       join(new_tries, plan[1:], tup)
```

Fig. 6. The Free Join algorithm.

3.3 Execution of the Free Join Plan

The execution of a Free Join plan has two phases: the build phase and the join phase. The build phase constructs the GHTs for the relations in the query, by calling the constructor method `new` on each relation with the appropriate schema. The join phase works over the GHTs to compute the join of the relations.

Build Phase. The build phase constructs a GHT for each relation (atom) $R_i(\boldsymbol{x}_i)$, as follows. If the plan partitions the atom into the subatoms $R_i(\boldsymbol{y}_0), R_i(\boldsymbol{y}_1), \ldots, R_i(\boldsymbol{y}_{\ell-1})$, then the schema of its GHT is the list $[\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{\ell-1}, []]$. Recall that the last level of a GHT is a vector instead of a hash map. As an optimization, if the last subatom $R_i(\boldsymbol{y}_{\ell-1})$ is the cover of its node, then we drop the last $[]$ from the schema, in other words, we construct a vector for the $\boldsymbol{y}_{\ell-1}$. After computing the schema for each relation, we call the constructor method `new` on each relation and its computed schema to build the GHTs.

Example 3.10. Consider the plan in Eq. (2) for the clover query $Q_{\clubsuit}$. The GHT schemas for R, S, and T are $[[x, a]]$, $[[x], [b]]$, and $[[x], [c]]$ respectively. Thus, R is a flat vector of tuples, and each of S and T is a hash map of vectors of values. Consider now the triangle query $Q_{\triangle}$ and the plan $[[R(x, y), S(y), T(x)], [S(z), T(z)]]$. The GHT schemas for R, S, T are $[[x, y]]$, $[[y], [z]]$, and $[[x], [z], []]$: in other words R is stored as a vector, S is a hash-map of vectors, and T is a hash-map of hash-maps of vectors.

Join Phase. The pseudo-code for the Free Join algorithm is shown in Figure 6. The `join` method takes as input the GHTs, the Free Join plan, and the current tuple initialized to be empty. If the plan is empty, we output the tuple (line 3). Otherwise, we work on the first node in the plan and intersect relevant tries (line 5). We iterate over tuples in the covering relation, which is the first trie in the node (line 7). Then, we use values from `t` and the `tuple` argument as keys to probe into the other tries (line 8-15). To construct a key for a certain trie, we find the values mapped from the trie's schema variables in `t` and `tuple` (line 12). If any probe fails, we continue to the next tuple in

```
R = GHT("R",[["x","a"]])
S = GHT("S",[["x"],["b"]])
T = GHT("T",[["x"],["c"]])
for (x, a) in R:
  s = S[x]?
  for b in s:
    t = T[x]?
    for c in t:
      output(x, a, b, c)
```

(a) Binary Free Join.

```
# same as the left
# ...
# ...
for (x, a) in R:
  s = S[x]?
  t = T[x]?
  for b in s:
    for c in t:
      output(x, a, b, c)
```

(b) Factorized Free Join.

```
R = GHT("R",[["x"],["a"]])
# same as the left
# ...
for x in R:
  r=R[x]?; s=S[x]?; t=T[x]?
  for a in r:
    for b in s:
      for c in t:
        output(x, a, b, c)
```

(c) Generic Free Join.

Fig. 7. Execution of Free Join for the clover query.

the outer loop. If all probes succeed, we replace the original tries with the subtries returned by the probes, and recursively call `join` on the new tries and the rest of the plan (line 16-17).

The recursive definition may obscure the essence of the Free Join algorithm, so we provide some examples where we unroll the recursion. We introduce some convenient syntax to simplify the presentation. We write `for (x,y,...) in T:` to introduce a for-loop iterating over `T`, binding the values of each tuple in `T.iter()` to the variables `x,y,...`. We write `r = R[t]?` to bind the result of `R.get(t)` to `r`; if the lookup fails, we continue to the next iteration of the enclosing loop. In other words, `r = R[t]?` is equivalent to:

```
r = R.get(t); if r.is_none(): continue
```

Example 3.11. Consider the plan in Eq. (2) for the clover query $Q_\clubsuit$. Figure 7a shows its execution; ignore the underlined instruction for now. In the build phase, we construct a flat vector for R and a hash table for each of S and T. In the join phase, for the node $[R(x,a), S(x)]$ we iterate over R and probe into S, while for the second node $[S(b), T(x)]$, we iterate over the second level of S and probe into T. Finally, the third loop iterates over the second level of T and outputs the result.

Example 3.12. Consider now the plan in Eq. (3) for $Q_\clubsuit$. Its execution is shown in Figure 7c. We construct hash tables for R, S, and T, keyed on x. The first loop level intersects the three relations on x, and subsequent loop levels take the Cartesian product of the relations on a, b, and c.

Note that Fig. 7a follows the execution of binary hash join with the plan $[R, S, T]$, whereas Fig. 7c follows the execution of Generic Join with the plan $[x, a, b, c]$. We will describe Fig. 7b later.

3.4 Discussion

Free Join plans generalize both traditional binary plans and Generic Join. One limitation so far is our assumption that the cover is chosen during the *build phase*. This was convenient for us to illustrate how to avoid constructing some hash maps, by storing the last level of a GHT as vector, when it corresponds to a cover. In contrast, Generic Join computes the intersection $R_1.x \cap R_2.x \cap \cdots$ by iterating over the smallest set, hence it chooses the "cover" at run time. We will address this in the next section by describing COLT, a data structure that constructs the GHT lazily, at run time, allowing us to choose the cover during the *join phase*.

4 OPTIMIZING THE FREE JOIN PLAN

In the previous section we have introduced Free Join plans and their associated data structures, the GHTs. We have seen that a Free Join plan is capable of covering the entire design space in Fig. 1,

```
fn binary2fj(bin_plan):
  fj_plan = []; r = bin_plan[0]
  φ0 = [ r(r.schema) ]; φ = φ0 # iterate over left relation
  for s in bin_plan[1:]:
    φ.push(s(s.schema ∩ avs(φ))) # probe w/ available vars
    fj_plan.push(φ)
    φ = [ s(s.schema - avs(φ)) ] # iterate over probe result
  fj_plan.push(φ)
  return fj_plan
```

Fig. 8. Translating a binary plans to a Free Join plan.

from traditional join plans to Generic Join. In this section we describe how to build, optimize, and speedup the execution of a Free Join plan. We start from a conventional binary plan produced by a query optimizer, and convert it into an optimized Free Join plan (Section 4.1). Next, we introduce the COLT data structure to greatly reduce the cost of building the hash tries (Section 4.2). We present a simple vectorized execution algorithm for Free Join (Section 4.3), and finally, we discuss how Free Join relates to Generic Join (Section 4.4).

4.1 Building and Optimizing a Free Join Plan

Our system starts from an optimized binary plan produced by a traditional cost-based optimizer; in particular, we use DuckDB's optimizer [24, 25]. We decompose a bushy plan into a set of left-deep plans, as described in Sec. 2, then convert each left-deep plan into an equivalent Free Join plan. Finally, we optimize the converted Free Join plan, resulting in a plan that can be anywhere between a left-deep plan or a Generic Join plan.

The conversion from a binary plan to an equivalent Free Join plan is done by the function `binary2fj` in Figure 8. We begin by adding the full atom of the left relation as the first subatom in the first Free Join plan node. Then we iterate over the remaining relations in the binary join plan. For each relation, we add a subatom whose variables are the intersection of the relation's schema with the available variables at the current Free Join plan node. Then we create a new join node, adding to it the relation with the remaining variables.

Example 4.1. The binary plan $[R, S, T]$ for the clover query $Q_\clubsuit$ is converted into the Free Join plan shown in Eq. (2). For another example, consider a chain query:

$$Q \text{ :- } R(x, y), S(y, z), T(z, u), W(u, v).$$

The left-deep plan $[R, S, T, W]$ is converted into:

$$[[R(x, y), S(y)], [S(z), T(z)], [T(u), W(u)], [W(v)]]$$

So far the algorithm in Figure 8 produces a Free Join plan that is equivalent to the left-deep plan. Next, we optimize the Free Join plan. The main idea behind our optimization is to bring the query plan closer to Generic Join, without sacrificing the benefits of binary join.

For intuition, let us revisit the clover query $Q_\clubsuit$, and its execution depicted in Fig. 7a (as explained in Example 3.11). Consider the input shown in Fig. 3b. Both relations R and S are skewed on the value x_2, and their join will produce n^2 tuples, namely $\{(x_2, a_i, b_j) \mid i, j \in [1..n]\}$. This means the body of the second loop in Figure 7a is executed n^2 times. However, the n^2 tuples are only to be discarded by the join with T which does not contain x_2.

```
fn factor(plan):
  @outer: for i in [1..n-1].reverse():
    φ = plan[i]; φ′ = plan[i-1]
    for α in φ:
      if α.vars ⊆ avs(φ) ∧ α.relation ∉ φ′:
        φ.remove(α); φ′.push(α)
      else: continue @outer
```

Fig. 9. Factorizing a Free Join plan.

There is a simple fix to the inefficiency: we can pull the underlined lookup on T in Figure 7a out of the loop over s to filter out redundant tuples early. This results in the nested loops in Figure 7b which runs in $O(n)$ time, because the two lookups in the first loop already filter the result to a single tuple. At the logical level, we convert the first Free Join plan into the second Free Join plan:

Naive plan (Eq. (2)):	$[[R(x,a),S(x)],[S(b),T(x)],[T(c)]]$
Optimized plan:	$[[R(x,a),S(x),T(x)],[S(b)],[T(c)]]$

While this is closer to the Generic Join in Figure 7c, it differs in that it still uses the same GHTs built for original plan, without the need for an additional hash table for R.

More generally, we will optimize a Free Join plan by *factoring out* lookups, i.e. by moving a subatom from a node Φ_i to the node Φ_{i-1}. In doing so we must ensure that the plan is still valid, and also avoid accidental slowdowns. For example, we cannot factor the lookups on S and T beyond the outermost loop, because that loop binds the variable x used in the lookups.

The optimization algorithm for Free Join plans is shown in Figure 9. We traverse the plan in reverse order visiting each node. For each node, if there is an atom whose variables are all available before that node, and if the previous node does not contain an atom of the same relation, we move the atom to the previous node. These two checks ensure the factored plan remains valid. The last line in the algorithm ensures we factor lookups *conservatively*. That is, we factor out a lookup only if all previous lookups in the same node have also been factored out. Doing so respects the lookup ordering given by the original cost-based optimizer, since scrambling this ordering may inadvertently slow down the query. It should be clear that, except for extreme cases where the enclosing loop is empty, factoring out any lookup will always improve performance.

4.2 COLT: Column-Oriented Lazy Trie

The original Generic Join algorithm builds a hash trie for each input relation. A left-deep plan avoids building a hash table on the left most relation, since it only needs to iterate over it, and this is an important optimization, since the left most relation is often the largest one. Building a subtrie can also be wasteful when that subtrie's parent is pruned away by an earlier join, in which case the subtrie will never be used. To address that, we describe here how to build the tries *lazily*: we only build the trie for a (sub-)relation at runtime, if and when we need to perform a lookup, or need to iterate over a prefix of its tuples. This idea leads to our new data structure called Column-Oriented Lazy Trie, or COLT for short. In our system the raw data is stored column-wise, in main memory, and each column is stored as a vector, as standard in column-oriented databases [1].

Definition 4.2. A *COLT* is a tree where each leaf is a vector of offsets into the base relation, and each internal node is a hash map mapping a tuple to a child node.

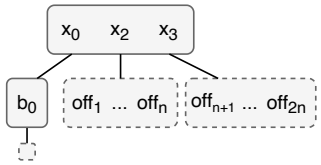


Fig. 10. A COLT for the relation S in Fig. 3a. Each off_i is an integer representing an offset into the table S.

```
struct COLT {
  relation, schema, vars,
  data = Map(HashMap<Tuple, COLT>) | Vec<Vec<u64>> }

impl GHT for COLT:
  fn new(relation, schema):
    COLT { relation, schema, schema[0],
           data = [ 0, 1, ..., relation.len - 1 ] }

  fn iter():
    match self.data:
      Map(m) => m.keys().iter(),
      Vec(v) =>
          if is_suffix(self.vars, relation.schema):
            v.map(|i| cols = self.relation[self.vars];
                      cols.map(|c| c[i]) )
          else: self.force(); self.iter()

  fn get(key): self.force(); self.get_map.get(key)

  fn force():
    match self.data:
      Map(m) => {} # already forced, do nothing
      Vec(v) =>
        map = new()
        for i in v:
          cols = self.relation[self.vars]
          k = cols.map(|col| col[i])
          if map[k] is None: # make a new, empty COLT
            map[k] = COLT { relation: self.relation,
                            schema: self.schema[1..],
                            data: [] }
          map[k].data.push(i)
        self.data = Map(map)
```

Fig. 11. The COLT data structure.

A COLT tree need not be balanced, and there can be both hash maps and vectors at the same tree level. Fig. 10 illustrates a COLT tree for the instance S of the clover query $Q_\clubsuit$.

COLT Implements the GHT interface in Figure 4, and its implementation is shown in Figure 11. As before, COLT stores a reference to the relation it represents, as well as the GHT schema computed from the plan. Consider a relation with n tuples. The COLT tree is initialized with a single node consisting of the vector $[0, \ldots, n-1]$, i.e. one offset to each tuple. COLT implements the `get` and `iter` methods lazily. When `get` is called, we check if the current node is a hash map or a vector. In the first case, we simply perform a lookup in the map. In the second case, we first replace the current vector with a hash map, whose children are vectors of offsets. Notice that this requires iterating over the current vector of offsets, accessing the tuple in the base table, inserting the key in the hash map, and inserting the offset in the corresponding child. Consider now a call to `iter`. If the current node is a hash map, then we return an iterator over it. If it is a vector, then we check if it is a suffix of the relation schema: if yes, then we simply iterate over that vector (and access the tuples via their offsets), otherwise we first materialize the current hash map as explained above, and return an iterator over the hash map.

As a simple but effective optimization, we do not initialize the COLT tree to the single node $[0, 1, \ldots, n-1]$, but instead iterate directly over the base table, if required. If no `get` is performed on this table, then we have completely eliminated the cost of building any auxiliary structure on this table. Thus, the Free Join plan can be equivalent to a left-deep plan that avoids building a hash table on the left-most relation. COLT is also closer to the structure of traditional hash tables, which, in some implementations, map a key to a vector of pointers to tuples.

Example 4.3. Consider an extension of the clover query $Q_\clubsuit$:

$$Q(x, a, b, c) \text{ :- } R(x, a), S(x, b), T(x, c), \underline{U(b)}.$$

Generic Join builds a 2-level hash trie for each of R, S, and T, as well as a 1-level hash trie for U. Consider the Free Join plan $[[R(x, a), S(x), T(x)], [U(b), S(b)], [T(c)]]$. Free Join executes the first node of the plan by iterating over R directly, without constructing any auxiliary structure. For each tuple (x, a) in R, it looks up x in S and T. Upon the first lookup, COLT builds the first level of the GHT for S and T, i.e. a hash map indexed by the x values. Assuming the database instance for R, S, T shown in Fig. 3a, the result of $R.x \cap S.x \cap T.x$ has only one value, x_0, thus, Free Join executes the second node for only one value x_0. Here it needs to intersect $U(b)$ and $S(b)$. Assume for the moment that Free Join chooses $U(b)$ to be the cover, on the first lookup in S, COLT will expand the second level, arriving at Figure 10: notice that all other b values in S will never be inserted in the hash table. More realistically, Free Join follows the principle in Generic Join and chooses $S(b)$ as cover, because it is the smallest: it builds a hash map for U, but none for the 2nd level of S.

The example highlights a divergence between Generic Join and traditional plans. To intersect $R_1.x \cap R_2.x \cap \ldots$, Generic Join choose to iterate over the smallest relation, which results in the best runtime *ignoring* the build time. A traditional join plan will iterate over the largest relation, because then it needs to build hash tables only on the smaller relations. Currently, we follow Generic Join, and plan to explore alternatives in the future.

4.3 Vectorized Execution

The Free Join algorithm as presented in Figure 6 suffers from poor temporal locality. In the body of the outer loop, we probe into the same set of relations for each tuple. However, these probes are interrupted by the recursive call at the end, which is itself a loop interrupted by further recursive calls.

A simple way to improve locality is to perform a batch of probes before recursing, just like the classic vectorized execution for binary join. Concretely, we replace the `iter` method with a new method `iter_batch(batch_size)` which returns up to `batch_size` tuples at a time. If there are less

```
@outer for ts in tries[0].iter_batch(batch_size):
  tup_subtries = [(tuple + t, [ tries[0].get(t) ]) | t ∈ ts]
  for trie in tries[1..]:
    for (tup, subtries) in tup_subtries:
      subtrie = trie.lookup(tup[trie.vars])
      if subtrie is None:
        tup_subtries.remove((tup, subtries))
      else: subtries.append(subtrie)
  for (tup, subtries) in tup_subtries:
    new_tries = all_tries[tries ↦ subtries]
    join(new_tries, plan[1:], tup)
```

Fig. 12. Vectorized execution for Free Join.

than `batch_size` tuples left, it returns all the remaining tuples. Then we replace the outer loop in Figure 6 with the one in Figure 12. For each batch of tuples, we create a vector pairing each tuple to its subtrie in `tries[0]`. Then for each trie to be probed, we iterate over the vector and look up each tuple from the trie. If the lookup succeeds, we append the subtrie to the vector of tries paired with the tuple. If it fails, we remove the tuple to avoid probing it again. Finally, with each tuple and the subtries it pairs with, we recursively call `join`.

4.4 Discussion

COLT is a lazy data structure, sharing a similar goal with database cracking [10, 11], where an index is constructed incrementally, by performing a little work during each lookup. Another connection is to Factorized Databases [22] – we intentionally used the term "factor" when describing how we optimize Free Join plans to suggest this connection. Concretely, we can view the trie data structure as a factorized representation of a relation, where keys of the same hash map are combined with union, and tuples are formed by taking the product of values at different levels. Practically, we can use this factorized representation to compress large outputs, saving time and space during materialization.

As we discussed at the end of Section 3, in order match the optimality of Generic Join, the Free Join algorithm needs to choose dynamically the "cover", i.e. the relation over which to iterate. To achieve this, we first find *all* covers for each node, then make a simple change to the Free Join algorithm in Figure 6: we simply choose to iterate over the cover whose trie has the fewest keys. For that we insert the following code right before the outer loop in Figure 6:

```
trie[0] = covers(plan[0]).min_by(|t| t.keys().len)
trie[1..] = # the rest of the tries
```

When we use COLTs, we cannot know the exact number of keys in a vector unless we force it into a hash map. In that case we use the length of the vector as an estimate.

Example 4.4. Consider the triangle query $Q_\triangle$, and the following Free Join plan:

$$[[R(x), T(x)], [R(y), S(y)], [S(z), T(z)]]$$

Each subatom is a cover of its own node. On the outermost loop, we iterate over R if it has fewer x values, and otherwise we iterate over T. On the second loop level we make a decision picking between S and a subtrie of R, *for each subtrie of* R. Finally, on the innermost loop we pick between the subtries of S and T.

5 EXPERIMENTS

We implemented Free Join as a standalone Rust library. The main entry point of the library is a function that takes a binary join plan (produced and optimized by DuckDB), and a set of input relations. The system converts the binary plan to a Free Join plan, optimizes it, then runs it using COLT and vectorized execution. We compare Free Join against two baselines: our own Generic Join implementation in Rust, and the binary hash join implemented in the state-of-art in-memory database DuckDB [24, 25]. We evaluate their performance on the popular Join Order Benchmark (JOB) [15] and the LSQB benchmark [16]. In addition, we compare against Kùzu [6], a system that implements Generic Join. Kùzu is the current iteration of the Graphflow system [17]. We ask three research questions:

(1) How does Free Join compare to binary and Generic Join, on acyclic and cyclic queries?
(2) What is the impact of COLT and vectorization on Free Join?
(3) How sensitive is Free Join to the query optimizer's quality?

5.1 Setup

While we had easy access to optimized join plans produced by DuckDB, we did not find any system that produces optimized Generic Join plans, or can take an optimized plan as input. We therefore implement a Generic Join baseline ourselves, by modifying Free Join to fully construct all tries, and removing vectorization. We chose as variable order for Generic Join the same as for Free Join.[4]

Both the JOB and the LSQB benchmarks focus on joins. JOB contains 113 acyclic queries with an average of 8 joins per query, whereas LSQB contains both cyclic and acyclic queries. Each query in the benchmarks only contains base-table filters, natural joins, and a simple group-by at the end, and no null values. JOB works over real-world data from the IMDB dataset, and LSQB uses synthetic data. We exclude 5 queries from JOB that return empty results, since such empty queries are known to introduce reproducibility issues[5]. We use the first 5 queries from LSQB; the other 4 queries require anti-joins or outer joins which we do not support.

We ran all our experiments on a MacBook Air laptop with Apple M1 chip and 16GB memory. All systems are configured to run single-threaded in main memory, and we leave all of DuckDB's configurations to be the default. All systems are given the same binary plan optimzed by DuckDB. To answer our third research question, we needed to hijack DuckDB's optimizer to produce a poor plan. We did this by modifying its cardinality estimator to always return 1. Since we are only interested in the performance of the join algorithm, we exclude the time spent in selection and aggregation when reporting performance. This excluded time takes up on average less than 1% of the total execution time.

5.2 Run time comparison

Our first set of experiments compare the performance of Free Join, Generic Join, and binary join on the JOB and LSQB benchmarks. For each query in the benchmarks, we invoke DuckDB to obtain an optimized binary plan, and provide the plan to our Free Join and Generic Join implementation. We run LSQB with the scaling factors 0.1, 0.3, 1, and 3, as some queries run out of memory with larger scaling factors.

Figure 13 compares the run time of Free Join and Generic Join against binary join on JOB queries. We see that almost all data points for Free Join are below the diagonal, indicating that Free Join is faster than binary join. On the other hand, the data points for Generic Join are largely above the diagonal, indicating that Generic Join is slower than both binary join and Free Join. On average

[4] Free Join defines only a partial order; we extended it to a total order.
[5] See GitHub issue: https://github.com/gregrahn/join-order-benchmark/issues/11

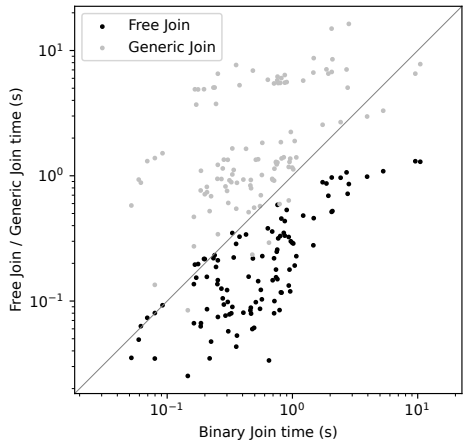


Fig. 13. Run time comparison on JOB. Each black dot compares the run time of a query on Free Join and binary join, and a black dot below the diagonal means Free Join is faster. The gray dots compare Generic Join and binary join similarly.

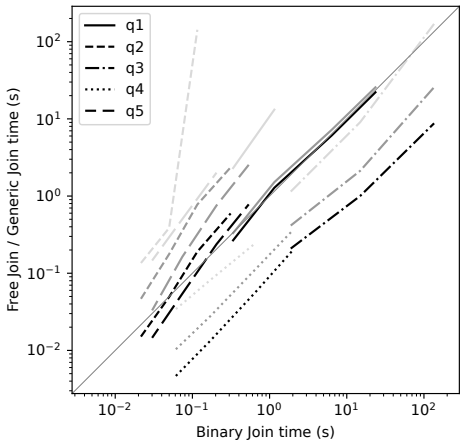


Fig. 14. Run time comparison on LSQB. Each line is a query running on increasing scaling factors (0.1, 0.3, 1, 3). The black lines compare Free Join with binary join, the gray lines compare our Generic Join baseline with binary join, and the light gray lines compare Kùzu with binary join.

(geometric mean), Free Join is 2.94x faster than binary join and 9.61x faster than Generic Join. The maximum speedups of Free Join against binary join and Generic Join are 19.36x and 31.6x, respectively, while the minimum speedups are 0.85x (17% slowdown) and 2.63x.

We zoom in onto a few interesting queries for a deeper look. The slowest query under DuckDB is Q13a, taking over 10 seconds to finish. Generic Join runs slightly faster, taking 7 seconds, whereas Free Join takes just over 1 second. The query plan for this query reveals the bottleneck for binary join: the first 3 binary joins are over 4 very large tables, and two of the joins are many-to-many joins, exploding the intermediate result to contain over 100 million tuples. However, all 3 joins are on the same attribute; in other words they are quite similar to our clover query $Q_{\clubsuit}$. As a result, Generic Join and Free Join simply intersects the relations on that join attribute, expanding the remaining attributes only after other more selective joins. This data point appears to confirm a folklore that claims WCOJ algorithms are more resistant to poor query plans. After all, binary join could have been faster, had the query plan ordered the more selective joins first. We expand on this point with more experiments evaluating each algorithm's robustness against poor plans in Section 5.4.

On a few queries Free Join runs slightly slower than binary join, as shown by the data points over the diagonal. The binary plans for these queries are all bushy, and each query materializes a large intermediate relation. We have not spent much effort optimizing for materialization, and we implement a simple strategy: for each intermediate that we need to materialize, we store the tuples containing all base-table attributes in a simple vector. Future work may explore more efficient materialization strategies, for example only materializing attributes that are needed by future joins.

Figure 14 compares the performance of Free Join and Generic Join against binary join on LSQB queries. Each line corresponds to one query running on scaling factors 0.1, 0.3, 1, and 3. The black lines are for Free Join, gray lines for our own Generic Join baseline, and light gray lines for Kùzu. Kùzu errors when loading data for SF3; it did not finish after 10 minutes for q1 SF 1. DuckDB also

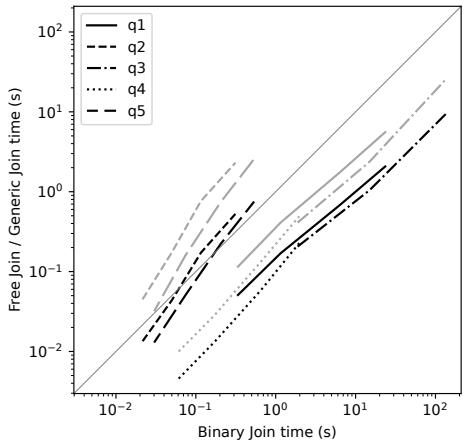


Fig. 15. Run time comparison on LSQB w/ factorization.

took over 10 minutes running q3 SF 3. These instances do not show up in the figure. We can see Kùzu takes consistently longer than our Generic Join implementation on all queries across scaling factors. This shows that our Generic Join implementation is a reasonable baseline to compare against. On cyclic queries, Free Join is up to 15.45x (q3) faster than binary join, and up to 4.08x (q2) faster than Generic Join. On acyclic queries Free Join is up to 13.07x (q4) and 3.25x (q5) faster than binary join and Generic Join, respectively. On q3 and q4 both Free Join and Generic Join consistently outperform binary join on all scaling factors. q3 contains many cycles, whereas q4 is a star query, so the superior performance of Free Join and Generic Join is expected. Surprisingly, despite q2 being a cyclic query, Free Join is only slightly faster on smaller inputs and is even slightly slower on larger inputs. This is the opposite of the common believe that WCOJ algorithms should be faster on cyclic queries. The query plan reveals that there are no skewed joins, and so binary join suffers no penalty. Our experience shows that, in practice, the superiority of WCOJ algorithms like Free Join and Generic Join is not solely determined by the cyclicity of the query; the presence of skew in the data is another important factor.

Unlike the JOB queries, in LSQB the output size (before aggregation) is much larger than the input size. This leads to a large amount of time spent in constructing the output, which involves random accesses to retrieve the data values for each tuple. We therefore implemented the optimization in Section 4.4 to factorize the output. This made q1 significant faster, as shown in Figure 15, while other queries are not affected.

5.3 Impact of COLT and Vectorization

The three key ingredients that make Free Join efficient are (1) our algorithm to optimize the Free Join plan by factoring, (2) the COLT data structure, and (3) the vectorized execution algorithm. We conduct an ablation study to evaluate the performance impact of these components. But if we do not optimize the Free Join plan converted from a binary plan and execute it as-is, Free Join would behave identically to binary join. Since we have already compared Free Join with binary join in Section 5.2, we do not repeat it here. Therefore, our ablation study includes two sets of experiments, evaluating the impact of COLT and vectorization respectively.

Figure 16 compares the run time of Free Join using different trie data structures. The baseline fully expands each trie ahead of time, and we call this *simple trie*. Another data structure, *simple lazy*

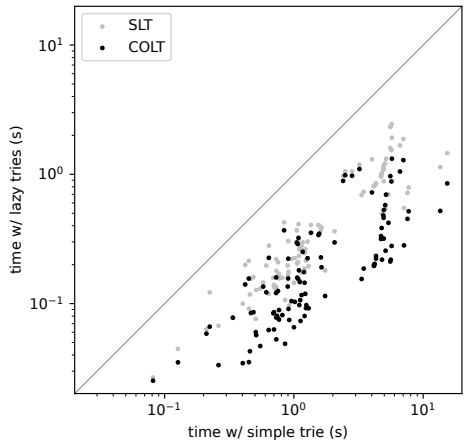


Fig. 16. Impact of COLT.

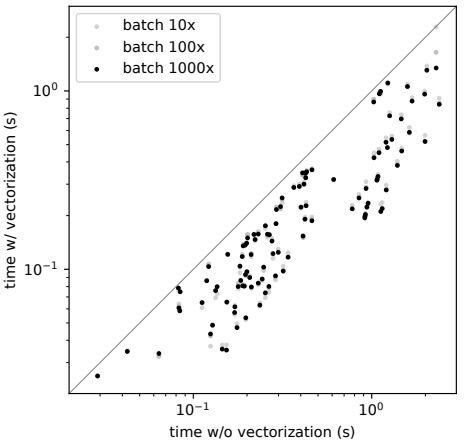


Fig. 17. Impact of vectorization.

trie (SLT), expands the first level of each trie ahead of time, while expanding the inner levels lazily. This is the same strategy as proposed by Freitag et al. [7]. Finally, COLT is our column-oriented lazy trie. In all three cases, we use the default vectorization batch size 1000. The experiments show the average (geometric mean) speedup of COLT is 1.91x and 8.47x, over SLT and simple trie respectively, and the maximum speedup over them is 11.01x and 26.29x, respectively.

Figure 17 compares the run time of Free Join using different vectorization batch sizes. The baseline uses no vectorization, i.e., we set the batch size to 1. Then we adjust the batch size among 10, 100, and 1000. The data does not show significant performance differences among the different batch sizes – it appears *any amount of vectorization is better than none*. For short-running queries, a smaller batch size perform slightly better, and for longer running queries a larger batch size wins. We conjecture this is due to a smaller batch having less overhead, leading to lower latency, while a larger batch size speeds up large joins better, leading to better throughput. Overall, using the default batch size 1000 leads to an average (geometric mean) speedup of 2.12x, and a maximum speedup of 5.33x over non-vectorized Free Join.

5.4 Robustness Against Poor Plans

Our last set of experiments compares Free Join, Generic Join and binary join on their sensitivity to the quality of the query plan. Many believe WCOJ algorithms suffer less from poor query planning, due to its asymptotic guarantees. Our experience with Q13 from Section 5.2 also seems to confirm this intuition. However, our experimental results tell a different story. As the first plot in Figure 18 shows, the relative performance of the three algorithms stays the same with good and bad plans, with Free Join being the fastest, Generic Join the slowest, and binary join in the middle. However, as shown in Figure 18, Free Join seems to slow down as much as binary join when the plan is bad (there are many points far above the diagonal). It turns out with a poor cardinality estimate, DuckDB routinely outputs bushy plans that materialize large results. We have noted in Section 5.2 that our materialization strategy is simplistic, so with larger intermediates it leads to more severe slowdown. In comparison, Generic Join slows down less (the data points are close to the diagonal). However, it was the slowest to begin with, and since overheads like trie building dominates Generic Join's run time, a bad plan does not make it much slower. Overall, we believe Free Join can be more robust to bad plans with faster materialization.

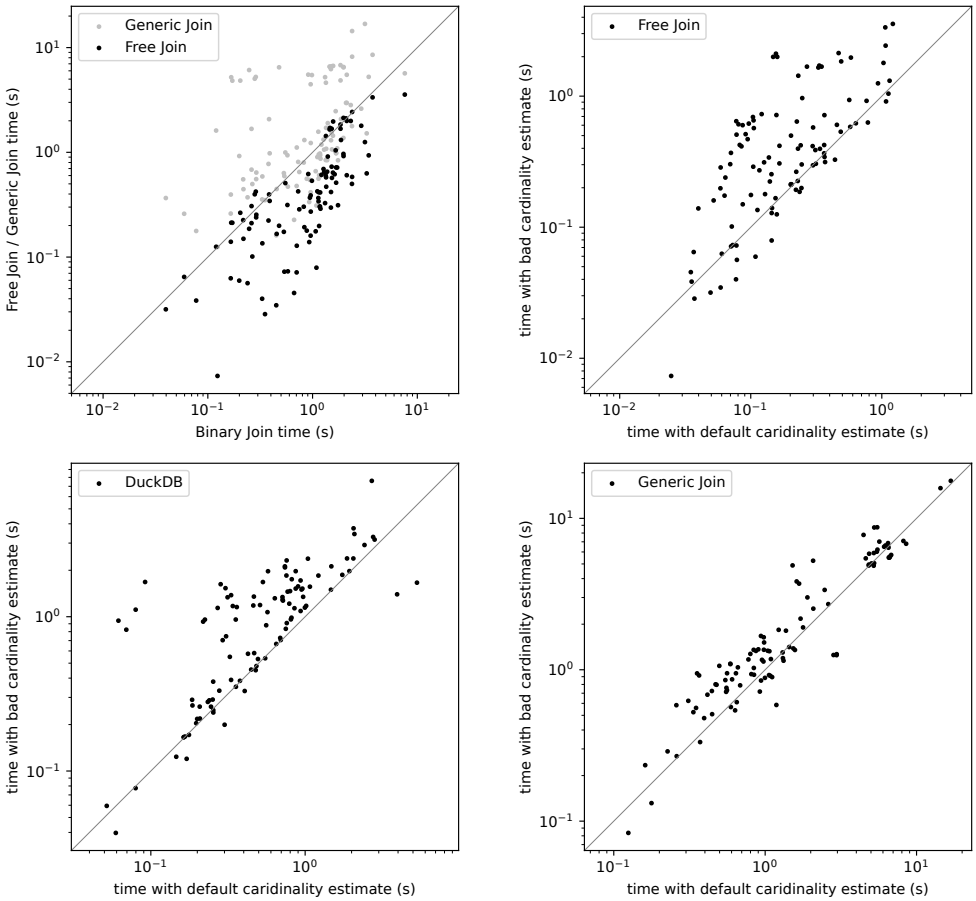


Fig. 18. Run time of join algorithms with good and bad cardinality estimates. The first plot compares the run time of Free Join, Generic Join and binary join on JOB with bad cardinality estimates. The remaining three plots compare the run time of each algorithm with good and bad cardinality estimates on JOB.

6 LIMITATIONS AND FUTURE WORK

With Free Join we have made a first step to bring together the worlds of binary join and WCOJ algorithms. There are three obvious limitations that require future work. Our current system is main memory only. If the data were to reside on disk, then COLT could be quite inefficient, because it requires repeated, random accesses to the data. Another limitation is that the optimization of a Free Join plan is split into two phases: a traditional cost-based optimization (currently done by DuckDB), followed by a heuristic-based optimization of the Free Join plan (factorization). This has the advantage of reusing an existing cost-based optimizer, but the disadvantage is that an integrated optimizer may be able to find better plans. Third, our current optimizer does not make use of existing indices. It is known that the optimization problem for join plans becomes harder in the presence of foreign key indices [15], and we expect the same to hold for Free Join plans. All these limitations require future work. As we have designed Free Join to closely capture binary join while also generalizing it, we hope the solutions to these problems can also be smoothly transferred from binary join to Free Join.

Finally, we have made several observations during this project, some of them quite surprising (to us), which we believe deserve a future study. We observed that a major bottleneck is the materialization of intermediate results in bushy plans; an improved materialization algorithm may speed up Free Join on bushy plans. One promising idea is to be more aggressively lazy and keep COLTs unexpanded during materialization, which essentially leads to a factorized representation of the intermediates. We also observed that, contrary to common belief, a cyclic query does not necessarily mean WCOJ algorithms are faster, and an acyclic query does not mean they are slow. A natural question is thus "when exactly are WCOJ algorithms faster than binary join?" Answering this question will also help us design a better optimizer for Free Join. The optimizer can output a plan closer to WCOJ when it expects major speedups. We note that the query optimizer by [7] switches between Generic Join and binary join depending on the estimated cardinality. In contrast, an optimizer for Free Join should smoothly transform a Free Join plan to fully explore the design space between the two extremes of binary join and Generic Join. Finally, we realized that, rather surprisingly, Generic Join and traditional joins diverge in their choice of the inner table (called the cover in our paper): Generic Join requires that to be the smallest (otherwise it is not optimal), while a traditional plan will chose it to be the largest (to save the cost of computing its hash table). Future work is required for a better informed decision for the choice of the inner relation.

ACKNOWLEDGMENTS

We thank Winston Bullen for his important contributions to experimental infrastructures of this project. This work is supported by NSF-BSF 2109922, NSF IIS 1954222, and NSF IIS 1907997.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[3] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440

[4] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 261–272. https://doi.org/10.1145/342009.335420

[5] Ronald Fagin. 1983. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM* 30, 3 (1983), 514–550. https://doi.org/10.1145/2402.322390

[6] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu Graph Database Management System. In *CIDR*.

[7] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[8] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170. https://doi.org/10.1145/152610.152611

[9] Goetz Graefe, Ross Bunker, and Shaun Cooper. 1998. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 86–97. http://www.vldb.org/conf/1998/p086.pdf

[10] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[11] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 413–424. https://doi.org/10.1145/1247480.1247527

[12] Alfons Kemper, Donald Kossmann, and Christian Wiesner. 1999. Generalised Hash Teams for Join and Group-by. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 30–41. http://www.vldb.org/conf/1999/P3.pdf

[13] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[14] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 429–444. https://doi.org/10.1145/3034786.3056105

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

[16] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021*, Vasiliki Kalavri and Nikolay Yakovets (Eds.). ACM, 8:1–8:11. https://doi.org/10.1145/3461837.3464516

[17] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

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

[19] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/3196959.3196990

[20] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini (Eds.). ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2213556.2213565

[21] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991

[22] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[23] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. 2001. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 567–574. https://doi.org/10.1109/ICDE.2001.914871

[24] Mark Raasveldt. 2022. DuckDB - A Modern Modular and Extensible Database System. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zait (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/Keynotes/Abstract_MarkRaasveldt.pdf

[25] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf

[26] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[27] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, Athens, Greece, 96–106. https://doi.org/10.5441/002/icdt.2014.13

[28] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

Received October 2022; revised January 2023; accepted February 2023