From Binary Join to Free Join

Yisu Remy Wang
UCLA
Los Angeles, CA, USA

Max Willsey
UC Berkeley
Berkeley, CA, USA

Dan Suciu
University of Washington
Seattle, WA, USA

ABSTRACT

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. In an effort to unify and generalize the two paradigms, we proposed a new framework, called Free Join, in our SIGMOD 2023 paper. Not only does Free Join unite the worlds of traditional and worst-case optimal join algorithms, it uncovers optimizations and evaluation strategies that outperform both.

In this article, we approach Free Join from the traditional perspective of binary joins, and re-derive the more general framework via a series of gradual transformations. We hope this perspective from the past can help practitioners better understand the Free Join framework, and find ways to incorporate some of the ideas into their own systems.

1. INTRODUCTION

Over the last decade, worst-case optimal join (WCOJ) algorithms [10, 14, 11, 9] 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 [11]. These algorithms opened up a flourishing field of research, leading to both theoretical results [11, 6] and practical implementations [14, 2, 4, 8].

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 [11], but when the query is acyclic, classic algorithms like the Yannakakis algorithm [16] 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.

The dichotomy of WCOJ versus binary join has led researchers and practitioners to view the algorithms as opposites. In our SIGMOD 2023 paper [15], we broke down this dichotomy with a new framework called Free Join that unifies WCOJ and binary join. Further more, we proposed new data structures, evaluation algorithms, and optimizations to make Free Join outperform both binary join and WCOJ.

In this article, we review Free Join from a new perspective: starting from the traditional binary join algorithm, we apply a series of gradual transformations to arrive at the Free Join algorithm as well as the WCOJ algorithm. We hope this perspective from the past can help practitioners better understand Free Join, and pave the way for its adoption into existing systems.

This article is based on the paper Free Join: Unifying Worst-case Optimal and Traditional Joins [15], published at SIGMOD 2023.

This is a minor revision of the paper entitled Free Join: Unifying Worst-Case Optimal and Traditional Joins, published in Proc. ACM Manag. Data, Vol. 1, No. 2, Article 150, June 2023.
2836-6573/2023/6-ART150 https://doi.org/10.1145/3589295

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

```
SELECT * FROM R, S, T WHERE R.x = S.x AND S.x = T.x
```

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

Figure 1: The clover query $Q_{\clubsuit}$, and an input instance. Note that x_0 is the only x-value in all three relations, therefore the only output tuple is (x_0,a_0,b_0,c_0).

2. FROM BINARY JOIN TO FREE JOIN

In this section we introduce the Free Join framework. Unlike our SIGMOD paper [15] which defines Free Join from the basic building blocks, here we start from the traditional binary join and gently massage it into the more general Free Join. To keep the presentation intuitive, we will be following an example instead of defining the algorithm in full generality. We refer the reader to our SIGMOD paper [15] for a more formal treatment.

2.1 Basic Concepts and Notations

For simplicity we consider only *natural join* queries, where all joins are equijoins, and all input relations are joined over common attributes. Such queries are also known as *conjunc-*

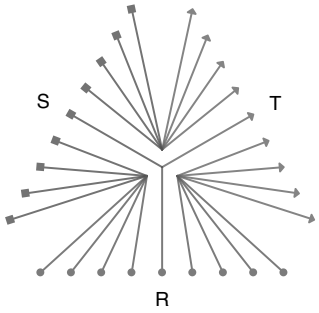

Figure 2: Visualization of the input relations to $Q_{\clubsuit}$. Each binary relation is represented as a set of edges on each side. The query $Q_{\clubsuit}$ looks for sets of 3 edges, one from each relation, that meet in the middle (there is only one such set, at the center of the figure).

```
for ...:
  m = M[x]?
  ...
```

```
for ...:
  if x not in M:
    continue
  else:
    m = M[x]
    ...
```

Figure 3: Example using the notation `m = M[x]?`. The two code fragments are equivalent.

tive queries and can be written in "Datalog notation" as the following example shows.

Example 1. *Consider SQL query in Figure 1. The corresponding conjunctive query appears above it, where each of $R(x,a)$, $S(x,b)$, and $T(x,c)$ is called a* body atom*, and $Q_{\clubsuit}(x,a,b,c)$ the* head atom.

It is often convenient to view a conjunctive query 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 body atoms in Q. The hypergraph for $Q_{\clubsuit}$ has four vertices, each for x, a, b, and c, and three edges, each for $R(x,a)$, $S(x,b)$, and $T(x,c)$. As standard, we say that the query Q is *acyclic* if its associated hypergraph is α-acyclic[1] [3]. Note that $Q_{\clubsuit}$ is acyclic, while an example of a *cyclic* query is the "triangle query":

$$Q_{\triangle}(x,y,z) \text{ :- } U(x,y), V(y,z), W(z,x).$$

whose query hypergraph is a triangle.

We now introduce a notation to make pseudocode cleaner. Inside a loop, we will write `m = M[x]?` for looking up x from the hash map M; if M contains x, we assign the result of the lookup to m; otherwise, we `continue` to the next iteration of the enclosing loop. In other words, the code fragments in Figure 3 are equivalent.

2.2 Binary Join

The standard approach to computing a natural join of multiple relations 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 R_i. 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 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. *A possible left-deep linear plan for $Q_{\clubsuit}$ is $[R,S,T]$, which represents $(R(x,a) \bowtie S(x,b)) \bowtie T(x,c)$. To execute this plan, we first build a hash table for S keyed on x, where each x maps to a vector of (x,a) tuples, and a hash table for T keyed on x, each mapped to a vector of (x,c) tuples*[2]*. Then the execution proceeds as shown in Figure 4a. For each tuple (x,a) in R, we first probe into the hash table for S using x to get a vector of (x,b) tuples. We then loop over each (x,a) and probe into the hash table for T with x. Each successful probe will return a vector of (x,c) tuples, and we output the tuple (x,a,b,c) for each (x,c). On the input instance in Figure 1 (visualized in Figure 2), this algorithm runs in time $\Omega(n^2)$.*

2.3 Columnar Storage and Late Materialization

The first transformation we perform on the binary join algorithm makes it work on column-wise storage instead of a row-wise one. This is not yet an optimization because it likely will not improve the performance, but this step serves as an important bridge to the next optimizations. As Figure 4b shows, in the outermost loop we iterate over row indices instead of tuples. For each row index i, we retrieve the x-value $R.x[i]$, as well as the corresponding a-value $R.a[i]$. The hash maps for S and T now map each x to a vector of row indices, so we next look up into the hash map for S using x to get a vector of j. In the second loop, we retrieve the x-value and b-value from S for each j, then probe into T to get a vector of k. Finally, we retrieve the x-value and c-value from T for each k, and output the tuple (x,a,b,c).

A key inefficiency of the algorithm in Figure 4b is that, although the query only outputs a single tuple (x_0,a_0,b_0,c_0),

[1]The reader does not need to be familiar with definitions of acyclic queries to understand Free Join.

[2]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 [5].

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

(a) Binary Free Join.

```
for i in 0..R.len():
  x = R.x[i]; a = R.a[i]
  s = S[x]? # s:x->[j]
  for j in s:
    x = S.x[j]; b = S.b[j]
    t = T[x]? # t:x->[k]
    for k in t:
      x = T.x[k]; c = T.c[k]
      output(x,a,b,c)
```

(b) Columnar storage.

```
for i in 0..R.len():
  x = R.x[i]; s = S[x]?
  for j in s:
    x = S.x[j]; t = T[x]?
    for k in t:
      x = T.x[k]
      a = R.a[i]
      b = S.b[j]
      c = T.c[k]
      output(x,a,b,c)
```

(c) Late materialization.

```
for i in 0..R.len():
  x = R.x[i];
  s = S[x]?; t = T[x]?
  for j in s:
    for k in t:
      a = R.a[i]
      b = S.b[j]
      c = T.c[k]
      output(x,a,b,c)
```

(d) Late iteration.

Figure 4: Execution of binary join for the clover query 4a, and three transformations. The first transformation 4b makes the algorithm work on column-wise storage instead of a row-wise one; the second transformation 4c performs the classic late materialization optimization; the last one is another transformation that we call late iteration 4d.

we still did a lot of work retrieving the different a, b, and c values from their respective columns. For example, since we iterate over the entire R relation, we retrieve all $2n + 1$ a-values from $R.a$; even worse, since $|R \bowtie S| = n^2 + 1$, we will access $S.b$ $\Omega(n^2)$ times. A better strategy is to *delay* the retrieval of these values until we actually need them. In this case, we can delay the retrieval of a, b, and c until we are ready to output the tuple (x, a, b, c). This way we only need to access each of $R.a$, $S.b$, and $T.c$ once, instead of $\Omega(n^2)$ times. This is precisely the classic *late materialization* optimization [1] now implemented in nearly all modern database systems.

2.4 Late Iteration and Free Join

We can go one step beyond late materialization and further optimize the code in Figure 4c. The key observation is that, although we retrieve x from the different relations in each loop level, *they have to be the same value* because x is the join attribute! This means we can remove the last two redundant retrievals of x and reuse the value from the outermost loop, which corresponds to removing the underlined code in Figure 4c. At this point, we can see that the remaining body of the second loop, `t = T[x]?`, does not depend on the loop variable j at all. We can therefore pull the lookup out of the loop, resulting in the code in Figure 4d. In other words, we *delay* the iteration over s until after the lookup on T succeeds.

Note that this final optimization has improved the asymptotic run time of the algorithm: although late materialization already saves a quadratic number of accesses to the relation columns, it still needs to iterate over $\Omega(n^2)$ row indices, because the first two loop levels essentially compute the join $R \bowtie S$. In contrast, the first loop level in Figure 4d joins every tuple of R with S and T at the same time, and the entire algorithm now runs in $O(n)$ time.

At the moment, our optimizations may appear rather low-level and ad-hoc. Taking a step back, we can understand the execution of any join algorithm as a series of iterations and lookups. The transformation from row-wise to column-wise storage involves changing *what to iterate over* (row indices instead of tuples); the late materialization optimization changes *what to look up* and *when to look up* (look up row indices first, then retrieve values later); finally, the late iteration optimization reorders the iterations and lookups.

While columnar storage and late materialization have become stables of modern database systems, the contribution of the Free Join framework is a new abstraction to describe the ordering of iterations and lookups that we call the Free Join plan. The basic building blocks of a Free Join plan are called *subatoms*, each of which is a subset of a relation schema.

Definition 1. *Given a relation schema $R(x_1, x_2, \ldots)$, a* subatom *is of the form $R(x_i, x_j, \ldots)$ where $\{x_i, x_j, \ldots\} \subseteq \{x_1, x_2, \ldots\}$.*

For example, given the relation R with schema $R(x, y)$, all of the following are valid subatoms: $R(), R(x), R(y), R(x, y)$. A Free Join plan over a set of schemas is a sequence of groups, where each group is a list of subatoms.

Definition 2. *Given a join query Q over $R_1, R_2, \ldots$, a Free Join plan for Q is of the form:*

$$[R_i(\mathbf{x}_i), R_j(\mathbf{x_j}), \ldots], [R_k(\mathbf{x_k}), \ldots], \ldots$$

where each of $R_i(\mathbf{x_i}), R_j(\mathbf{x_j}), R_k(\mathbf{x_k}), \ldots$ is a subatom over the schema of $R_i, R_j, R_k, \ldots$ respectively.

Each group in a Free Join plan corresponds to a loop level. At each loop level, we iterate over tuples of the first subatom in the group, and use the values to look up into the remaining subatoms. For this reason, we will sometimes stylize a Free Join plan as follows to emphasize the iterated subatom and reflect the loop nesting:

$$\begin{aligned}&[R_1(\mathbf{x_1}) \mid R_2(\mathbf{x_2}), R_3(\mathbf{x_3}), \ldots]\\ &\rightarrow [R_4(\mathbf{x_4}) \mid R_5(\mathbf{x_2}), \ldots]\\ &\rightarrow [R_6(\mathbf{x_6}) \mid \ldots]\end{aligned}$$

Example 3. *The Free Join plan for the algorithm in Figure 4d is:*

$$\begin{aligned}&[R(x, a) \mid S(x), T(x)]\\ &\rightarrow [S(b) \mid]\\ &\rightarrow [T(c) \mid]\end{aligned}$$

Although we only retrieve the value of a in the innermost loop, each a-value one-to-one corresponds to each i, so the plan iterates over both x and a at the first level.

Example 4. *We can represent the binary join algorithm in Figure 4a with the Free Join plan:*

$$\begin{aligned}&[R(x, a) \mid S(x)]\\ &\rightarrow [S(b) \mid T(x)]\\ &\rightarrow [T(c) \mid]\end{aligned}$$

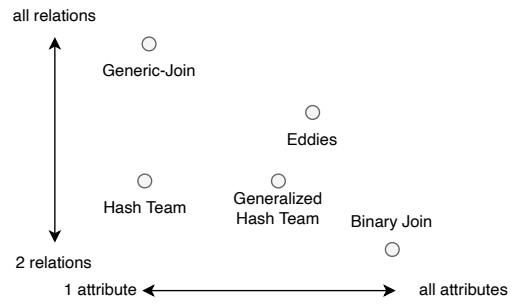

Figure 5: Free Join plans represent a design space of join algorithms that contains many existing algorithms.

if we ignore the redundant x at the inner loop levels.

In fact, every left-linear binary join plan $[R_1, R_2, R_3, \ldots]$ can be represented by a Free Join plan:

$$\begin{aligned}&[R_1(\mathbf{x_1}) \,|R_2(\mathbf{x_1} \cap \mathbf{x_2})]\\ \rightarrow\ &[R_2(\,\mathbf{x_2} \setminus \mathbf{x_1}) \mid R_3(\mathbf{x_2} \cap \mathbf{x_3})]\\ \rightarrow\ &[R_3(\mathbf{x_3} \setminus \mathbf{x_2}) \mid R_4(\mathbf{x_3} \cap \mathbf{x_4})]\\ &\vdots\end{aligned}$$

As we will see in Section 2.5, we can also represent any Generic Join plan with a Free Join plan. Free Join therefore generalizes and unifies both binary join and Generic Join. However, the true power of Free Join is its ability to represent algorithms like the one in Figure 4d that are neither binary join nor Generic Join. Intuitively, a (linear) binary join plan says which *relation* to process at each step, and always processes one additional relation at a time. A Generic Join plan says which *variable* to process at each step, and always processes one variable at a time. A Free Join plan can process any number of relations and variables at a time. As we show in Figure 5, this flexibilty allows Free Join to represent a much larger space of algorithms, leading to performance improvements beyond the existing algorithms.

2.5 Other Optimizations

In this section, we summarize a few additional optimizations introduced in our SIGMOD paper [15], as well as relate Free Join to the Generic Join algorithm. To motivate these optimizations, we will follow the new example query $Q_\triangle$ in Figure 7, and visualize an input instance in Figure 8. Note $Q_\triangle$ is now a *cyclic* query, because its hypergraph is a triangle with three vertices x, y, and z and three edges corresponding to $R(x, y)$, $S(y, z)$, and $T(z, x)$.

Let us first consider the algorithm in Figure 6a. We show the Free Join plan in the comments atop the figure, and note that it is equivalent to binary join. To reduce clutter we will stick with a row-wise notation while keeping in mind the underlying columnar storage. We also remove redudant values from the hash maps; for example, the hash map for S in Figure 6a now maps every y to a vector of z (instead of a vector of (y, z)). The binary join algorithm first iterates over (x, y)-tuples in R, using each y to probe into S to get a vector of z. For each z, it then probes into T to check if (z, x) is in T, and outputs the tuple (x, y, z) if so. This binary plan essentially computes the join $R \bowtie S$ first before discarding tuples that do not join with T. Because $R \bowtie T$ has size $\Omega(n^2)$, the algorithm runs in quadratic time. Since the relations are symmetric, any binary join plan will have the same asymptotic run time.

A different algorithm is shown in Figure 6b. Here, as we iterate over tuples in R, we look up into S and T at the same time. In other words we perform the late iteration optimization again, pulling up lookups to discard tuples early. However, this is not sufficient, because every tuple in R does in fact join with both S and T, so the first loop level discards no tuples. As the second loop level iterates over s, we are in effect still computing the join $R \bowtie S$ which takes $\Omega(n^2)$ time. To overcome this inefficiency, we now introduce a new operator called *intersection* ($\cap$).

2.5.1 Intersection

In contrast, the algorithm in Figure 6c runs in linear time. The small difference is that we have replaced the inner loop of Figure 6b, which iterates over s, with a loop iterating over the intersection of s and t. When we compute the intersection, we always iterate over the smaller set while probing into the larger set. To analyze the run time of this algorithm, we first assume we have built a hash map for S, mapping each y to a *hash set* of z, and similar for T. Building each hash map takes linear time. Then, as we iterate over R, we consider three cases:

1. For tuple (x_0, y_0), $s = S[y_0] = \{z_i \mid i \in [0, \ldots, n]\} = T[x_0] = t = s \cap t$, so we may iterate over either s or t to compute $s \cap t$ in linear time.

2. For each tuple $(x_0, y_i) \mid i > 0$, $t = \{z_i \mid i \in [0, \ldots, n]\}$ but $s = S[y_i] = \{z_0\}$, so we iterate over (the only element of) s and probe into t to compute $s \cap t$. This takes constant time for each y_i, so for all y_i the total time is linear.

3. The case for tuple (x_i, y_0) is symmetric to the previous case, so it also takes linear time.

Overall, the algorithm in Figure 6c runs in linear time. Another way to think about the intersection operation is to understand it as dynamically reording the iterations and lookups: to compute $s \cap t$, we switch between the plans $[S(z) \mid T(z)]$ and $[T(z) \mid S(z)]$ depending on which one is smaller.

2.5.2 Generic Join

We can now faithfully derive the Generic Join algorithm as a special case of Free Join, using the itersection operator. Suppose an instance of Generic Join follows the variable order $x_1, x_2, \ldots, x_n$. The corresponding Free Join plan is:

$$\begin{aligned}&[R_1^1(x_1) \cap R_1^2(x_1) \cap \cdots]\\ \rightarrow\ &[R_2^1(x_2) \cap R_2^2(x_2) \cap \cdots]\\ &\vdots\\ &\rightarrow [R_n^1(x_n) \cap R_n^2(x_n) \cap \cdots]\end{aligned}$$

where R_i^j is a relation that contains x_i in its schema. When computing a multiway intersection, we (dynamically) pick the smallest set to iterate over and probe into the rest. For any choice of the variable order, the Generic Join algorithm is guaranteed to run in worst-case optimal time [14, 9, 10].

```
# [ R(x, y) | S(y) ]
# -> [ S(z) | T(z, x) ]

for (x,y) in R:
  s = S[y]? # S:y->[z]
  for z in s:
    if (z,x) in T:
      output(x,y,z)
```

(a) Plan equivalent to binary join.

```
# [ R(x, y) | S(y), T(x) ]
# -> [ S(z) | T(z) ]

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

(b) Another Free Join plan.

```
# [ R(x, y) | S(y), T(x) ]
# -> [ S(z) ∩ T(z) ]

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

(c) Plan with intersect (∩).

```
# [ R(y) ∩ S(y) ]
# -> [ S(z) ∩ T(z) ]
#    -> [ T(x) ∩ R(x) ]
for y in R.y ∩ S.y:
  for z in S[y] ∩ T.z:
    for x in T[z] ∩ R[y]:
      output(x,y,z)
```

(d) Plan equivalent to Generic Join.

Figure 6: Four different Free Join plans for $Q_\triangle$ and their execution.

$$Q_\triangle \coloneq R(x,y), S(y,z), T(z,x).$$

```
SELECT * FROM R,S,T -- R(x,y), S(y,z), T(z,x)
 WHERE R.y = S.y AND S.z = T.z AND T.x = R.x
```

$$R = \{(x_0, y_i) \mid y \in [0 \dots n]\} \cup \{(x_i, y_0) \mid y \in [0 \dots n]\}$$
$$S = \{(y_0, z_i) \mid y \in [0 \dots n]\} \cup \{(y_i, z_0) \mid y \in [0 \dots n]\}$$
$$T = \{(z_0, x_i) \mid y \in [0 \dots n]\} \cup \{(z_i, x_0) \mid y \in [0 \dots n]\}$$

Figure 7: The triangle query $Q_\triangle$, and an input instance.

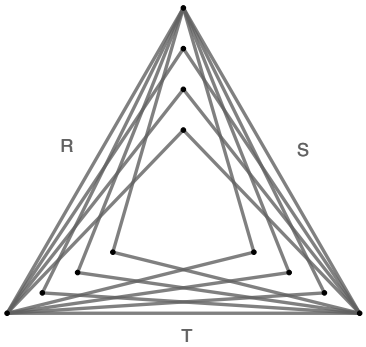

Figure 8: Visualization of input relations to $Q_\triangle$. Each relation is represented by a set of edges. The query $Q_\triangle$ looks for triangles formed by one edge from each relation (there are 10).

2.5.3 Lazy trie building

To explain the algorithm in Figure 6b we assumed to have pre-built hash maps for S and T. A more efficient strategy is to *lazily* construct parts of the data structures as we iterate over the relations. Specifically, we will only build the hash set for s (or t) right before we need to probe into it. This way, we can avoid building a linear number of (singleton) hash sets that we only need to iterate over. In the full paper [15] we describe a data structure, called Column-oriented Lazy Trie (COLT), that generalizes this idea.

2.5.4 Vectorized Execution

A simple way to implement Free Join is to use a recursive function, as shown in Figure 9. For every group in the Free Join plan, we iterate over the first subatom and probe into the remaining subatoms. If all probes are successful, we append new values to the partial tuple, and recursively call `join` on the remaining plan and sub-relations. This naïve implementation suffers from poor temporal locality: in the body of the loop, we probe into the same set of relations for each tuple. But these probes are interrupted by the recursive call at the end, which is itself a loop interrupted by further recursive calls.

```
def join(plan, tuple, R, S, T, ...):
  if plan is empty: output(tuple)
  else:
    let [ R(xs) | S(ys), T(zs), ... ] = plan[0]
    for xs in R:
      r = R[xs]?; s = S[ys]?; t = T[zs]?; ...
      join(plan[1:], tuple ++ xs, r, s, t, ...)
```

Figure 9: Recursive formulation of Free Join.

```
def join(plan, tuple, R, S, T, ...):
  if plan is empty: output(tuple)
  else:
    let [ R(xs) | S(ys), T(zs), ... ] = plan[0]
    tup_rels = {} # map a partial tuple to sub-relations
    for xs_batch in R.iter_batch():
      for xs in xs_batch:
        r = R[xs]?; s = S[ys]?; t = T[zs]?; ...
        tup = tuple ++ xs
        tup_rels[tup] = (r, s, t, ...)
    for (tup, rels) in tup_rels:
      join(plan[1:], tup, rels)
```

Figure 10: Vectorized execution for Free Join.

A simple way to improve locality is to perform a batch of probes before recursing, just like the classic vectorized execution for binary join. As shown in Figure 10, we call `iter_batch` to retrieve a batch of tuples from R. For each tuple in a batch, we probe into the relations to get the corresponding sub-relations. If all probes are successful, we append new values to the partial tuple, and pair the tuple with the respecitve sub-relations; otherwise we continue onto the next tuple in the batch. Finally, for each tuple that successfully probes into all relations, we call `join` recursively on the remaining plan.

3. 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

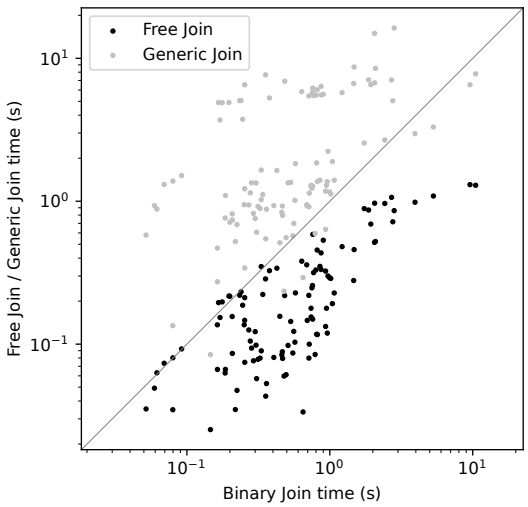

Figure 11: Run time comparison on JOB. Each black dot compares the run time of a query on Free Join and binary join, and a black dot below the diagonal means Free Join is faster. The gray dots compare Generic Join and binary join similarly.

in Rust, and the binary hash join implemented in the state-of-art in-memory database DuckDB [13, 12]. We evaluate their performance on the popular Join Order Benchmark (JOB) [7]. We refer the reader to our SIGMOD paper [15] for additional experiments that include other systems and benchmarks, as well as detailed ablation studies of the optimizations.

3.1 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.[3]

The JOB benchmark contains 113 acyclic join queries with an average of 8 joins per query. Each query in the benchmarks only contains base-table filters, natural joins, and a simple group-by at the end, and no null values. The queries works over real-world data from the IMDB dataset. We exclude 5 queries that return empty results, since such empty queries are known to introduce reproducibility issues[4].

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. 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.

[3]Free Join defines only a partial order; we extended it to a total order.

[4]See GitHub Issue #11: https://github.com/gregrahn/join-order-benchmark/issues/11

```
111101
JOIN
 |  \________9775
1919495      company_name(cid)
JOIN
 |  \________1
8123586      company_type(ctid)
JOIN
 |  \________1
24740873     info_type1(iid1)
JOIN
 |  \________1
148621556    info_type2(iid2)
JOIN
 |  \________1
177388547    kind_type(kid)
JOIN
 |  \________2609129
20885030     movie_companies(mid, cid, ctid)
JOIN
 |  \________1380035
 |            JOIN
 |             |  \_____1380035
 |            2528312   movie_info_idx(mid, iid1)
 |            title(mid, kid)
14835720
movie_info(mid, iid2)
```

Figure 12: Query plan produced by DuckDB for JOB Q13a. We show the join attributes next to each input relation, and label each relation with its size as well as each join with its (actual) cardinality.

3.2 Run time comparison

Figure 11 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 (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, as shown in Figure 12, 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 (`mid`); 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.

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 vector. Future work may explore more efficient materialization strategies, for example only materializing attributes that are needed by future joins.

4. CONCLUSION

In this paper we review the Free Join framework, which generalizes and unifies traditional join algorithms and WCOJ algorithms. We re-derive the more general Free Join from the well-understood binary join, hoping to make the framework more accessible to database practitioners. We hope to see the adoption of Free Join in mainstream databases, which shall inspire further research on the design of join algorithms. We conclude by pointing out some promising research directions. First, we have been focusing on single-threaded in-memory algorithms. How can we adapt Free Join to work on disk, on multi-core machines, and in distributed settings? In particular, the COLT data structure relies on laziness and appears inherently sequential. Do we need a new data structure to parallelize Free Join? Second, our optimizer starts from an already optimized binary plan, and conservatively improve it into a Free Join plan. How can we design and implement an optimizer to better exploit the flexibilty of Free Join? Finally, as the experiments show, our current implementation of Free Join is not yet competitive for certain bushy plans. How can we improve the performance of materializing intermediate results for Free Join?

5. REFERENCES

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE Computer Society, 2007.

[2] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[3] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[4] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[5] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[6] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 429–444. ACM, 2017.

[7] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[8] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[9] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124, New York, NY, USA, 2018. ACM.

[10] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48, New York, NY, USA, 2012. ACM.

[11] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

[12] M. Raasveldt. Duckdb - A modern modular and extensible database system. In S. R. Valluri and M. Zait, editors, *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, 2022.

[13] M. Raasveldt and H. Mühleisen. 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, 2020.

[14] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In N. Schweikardt, V. Christophides, and V. Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106, Athens, Greece, 2014. OpenProceedings.org.

[15] Y. R. Wang, M. Willsey, and D. Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[16] M. Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.