# TreeTracker Join: Simple, Optimal, Fast

ANONYMOUS AUTHOR(S)

We present a novel linear-time acyclic join algorithm, TreeTracker Join (TTJ). The algorithm can be understood as the pipelined binary hash join with a simple twist: upon a hash lookup failure, TTJ resets execution to the binding of the tuple causing the failure, and removes the offending tuple from its relation. Compared to the best known linear-time acyclic join algorithm, Yannakakis's algorithm, TTJ shares the same asymptotic complexity while imposing lower overhead. Further, we prove that when measuring query performance by counting the number of hash probes, TTJ will match or outperform binary hash join on the same plan. This property holds independently of the plan and independently of acyclicity. We are able to extend our theoretical results to cyclic queries by introducing a new hypergraph decomposition method called tree convolution. Tree convolution iteratively identifies and contracts acyclic subgraphs of the query hypergraph. The method avoids redundant calculations associated with tree decomposition and may be of independent interest. Empirical results on TPC-H, the Join Order Benchmark, and the Star Schema Benchmark demonstrate favorable results.

**ACM Reference Format:**
Anonymous Author(s). 2025. TreeTracker Join: Simple, Optimal, Fast. 1, 1 (December 2025), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Yannakakis [22] was the first to describe a linear-time join algorithm (hereafter YA) running in time $O(|\mathsf{IN}| + |\mathsf{OUT}|)$, where $|\mathsf{IN}|$ is the input size and $|\mathsf{OUT}|$ is the output size. In principle, this is the best asymptotic complexity one can hope for, because in most cases the algorithm must read the entire input and write the entire output. However, virtually no modern database systems implement YA. A major factor is its high overhead. Prior to executing the join, YA performs two passes over the input relations, using semijoins to reduce the input size. The reduction is lossless and enables optimally joining the reduced relations. Since the cost of a semijoin is proportional to the size of its arguments, this immediately incurs a 2× overhead in the input size. An improved version of YA [2] achieves the same result in one semijoin pass, but the overhead of this pass remains. Another practical challenge is that YA is "too different" from traditional binary join algorithms, making it difficult to integrate into existing systems. For example, the efficiency of YA critically depends on a query's *join tree* which is different from the query plan used by binary joins[1]. Where there is a wealth of techniques to optimize query plans for binary joins, little is known about cost-based optimization of join trees for YA.

In this paper, we propose a new linear-time join algorithm called TreeTracker Join (TTJ), inspired by the TreeTracker algorithm [10] in Constraint Satisfaction. TTJ can be understood as the traditional binary hash join with a twist: when a hash lookup fails, backtrack to the tuple causing the failure, and remove that tuple from its relation. The backtracking points depend only on the query, not the data, and are determined by the query compiler prior to query execution. The execution deviates

[1]By *join tree* we mean (hyper-)tree decomposition of hypertree width 1, not the tree of binary join operators commonly seen in relational algebra query plans.


ACM XXXX-XXXX/2025/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1 for i,x in R:
2   for y,j in S[x]:
3     for k in T[y]:
4       # lines left blank
5       # intentionally
6       for l in U[y]:
7         print(x,y,i,j,k,l)
```

(a) Binary join

```
1 for i,x in R:
2   for y,j in S[x]:
3     for k in T[y]:
4       if U[y] is None:
5         break
6       for l in U[y]:
7         print(x,y,i,j,k,l)
```

(b) Backjumping

```
1 for i,x in R:
2   for y,j in S[x]:
3     for k in T[y]:
4       if U[y] is None:
5         S[x].del((y,j)); break
6       for l in U[y]:
7         print(x,y,i,j,k,l)
```

(c) Tuple deletion

Fig. 1. Instantiation of binary hash join on example 1.1, with backjumping, and with tuple deletion.

from binary hash join only when a dangling tuple is detected and deleted. Thus the tuple is excluded from any computation going forward. Hence, when using identical query plans TTJ is guaranteed to match or outperform binary hash join (Section 4.2).

The following example illustrates the main ideas of TTJ.

*Example 1.1.* Consider the natural join of the relations $R(i,x)$, $S(x,y,j)$, $T(y,k)$, and $U(y,l)$, where we use $R(i,x)$ to denote that the schema of $R$ is $\{i,x\}$. The set $\{1,\ldots,N\}$ is denoted by $[N]$. Let the relations be defined as follows:

$$R = \{(i,1) \mid i \in [N]\} \quad S = \{(1,1,j) \mid j \in [N]\} \quad T = \{(1,k) \mid k \in [N]\} \quad U = \{(0,l) \mid l \in [N]\}$$

Observe $U$ shares no common $y$-values with $S$ or $T$, making the query result empty. We'll first consider execution with binary hash join, the foundation of our algorithm. Assume the optimizer produces a left-deep join plan $((R \bowtie S) \bowtie T) \bowtie U$. The execution engine builds hash tables for $S$, $T$, and $U$, mapping each $x$ to $(y,j)$ values in $S$, $y$ to $k$ values in $T$, and $y$ to $l$ values in $U$. Figure 1a[2] illustrates the basis of the execution. For each $(i,x)$ tuple in $R$, the hash table for $S$ is probed for the $(y,j)$ values. T is probed with each pair $(y,j)$ to determine the $k$ values. This repeats for $U$. Although the query produces no output, the execution takes $\Omega(N^3)$ time because it first computes the join of $R$, $S$, and $T$. A closer look at the execution reveals the culprit: when the lookup on $U$ produces no result, (line 6), the algorithm continues to the next iteration of the loop over $k$ values, (line 3). The same value of $y$ is used to probe into $U$ again! To address this, **the first key idea of TTJ is to backjump[3] to the level causing the probe failure.** For clarity we use `break` to represent the backjump, as shown in Figure 1b. When probing $U$ with the key value $y$ fails to return a result, we break out of the current loop over $k$ and continue to the next iteration of the second loop level. That is because the unsuccessful lookup key value of $y$ is assigned at that level. That next iteration retrieves new $y$, $j$ values, skipping over iterations for $k$ values that are doomed to fail. With this optimization, the execution finishes in $O(N^2)$ time, as it still needs to compute the join of $R$ and $S$. To improve the performance further: **the second key idea of TTJ is to delete the tuple causing the probe failure.** This is shown in Figure 1c: after the probe failure, the offending tuple $(x,y,j)$ is removed from the $S$ hash table. This is safe to do, because that $y$ value will always fail to join with any tuple in $U$. In this way, all tuples from $S$ are removed after looping over it the first time. Then, on all subsequent iterations of the loop over $R$, the probe into $S$ fails immediately. Overall, the algorithm finishes in $O(N)$ time.

[2]One may also recognize this as indexed nested loop join, which is equivalent [17].

[3]Backjumping is a concept in backtracking search algorithms; we use the term informally to mean the interruption of a nested loop iteration to jump back to an outer loop, while referring to the original TreeTracker algorithm [10] for a precise definition.

In general, TTJ runs in linear time in the size of the input and output for full acyclic joins. But the algorithm is not limited to acyclic queries: given the same query plan for *any* query, cyclic or acyclic, TTJ is guaranteed to match the performance of binary hash join, when measuring query performance by counting the number of hash probes. In particular, when no probe fails TTJ behaves identically to binary join. This is in contrast to YA which always carries the overhead of semijoin reduction, even if the reduction does not remove any tuple.

To address cyclic queries further we introduce a new method to break down a cyclic query into acyclic parts called *tree convolution*. We use this method to analyze the run time of TTJ. A special kind of tree convolution, called *rooted convolution*, eliminates materialization of intermediates during query processing.

In summary, our contributions include:

- Propose TTJ, a new join algorithm that runs in time $O(|\text{IN}| + |\text{OUT}|)$ on full acyclic queries.
- Prove that TTJ matches or outperforms binary join given the same query plan, on both acyclic and cyclic queries.
- Introduce *tree convolution*, a new method to break down cyclic queries into acyclic parts, and use it to analyze the run time of TTJ on cyclic queries.
- Improve the performance of TTJ with further optimizations.
- Conduct experiments to evaluate the efficiency of TTJ on acyclic queries.

## 2 Related Work

The observation that only one semijoin pass is necessary in YA has been a folklore in the database community, with an early appearance in the theoretical work of Bagan, Durand, and Grandjean [3]. Their paper studies the problem of enumerating conjunctive query results with constant delay, but without considering practical efficiency. Recent systems implementing such enumeration algorithms take advantage of the same insight [19, 20]. Compared to this improved version of YA, TTJ has the guarantee of matching the performance of HJ given any query plan, and is often faster in practice, as we will show in Section 6.

Researchers have also explored ways to integrate elements of YA into existing systems. Zhu et.al. [24] propose *lookahead information passing*, using bloom filters to implement semijoins over star schemas. Birler, Kemper, and Neumann [5] decompose every join operator into a *lookup* and an *expand*, and prove that certain lookup-and-expand (L&E) plans are guaranteed to run in linear time for acyclic queries. Bekkers et.al. [4] implement L&E plans in a vectorized query engine, while proving that their approach is guaranteed to match binary hash join for a class of *well-behaved* query plans. The theoretical guarantees of TTJ is complementary to these approaches: while TTJ guarantees to match binary hash join for left-deep plans, the well-behaved class defined by Bekkers et.al. [4] essentially contains right-deep plans with a slight generalization. On the other hand, as we focus on an algorithm-level evaluation of TTJ in this paper, our implementation is not yet competitive with the highly optimized systems mentioned above. Future work shall explore how to incorporate various system-level optimizations like query compilation, vectorization, and parallelization into TTJ.

Going beyond acyclic queries, the standard way to handle cyclic queries is to break up the query with (hyper-)tree decomposition [8]. Such decomposition results in smaller cyclic subqueries connected by an acyclic "skeleton". Each cyclic subquery can then be computed with worst-case optimal join algorithms [15, 21]. With the result of each subquery materialized, the final output can then be computed with YA. As we will show in Section 7, TTJ can support cyclic queries with only a few modifications. Compared to the tree decomposition approach, TTJ does not require materializing intermediate results, thus requiring only constant space in addition to the linear

space required to store and index the input relations. While the worst-case time complexity of TTJ does not match that obtained by tree decompositions, the advantage of each approach depends on the data.

As the name suggests, TreeTracker Join is a direct decendent of the TreeTracker algorithm [10] from Constraint Satisfaction. The TreeTracker CSP algorithm resolved Dechter's conjecture [6] that there existed an optimal algorithm for acyclic CSPs free of any preprocessing. The connection between query answering and constraint satisfaction is a recurring theme in the literature to the extent that an expression emerged, the problems are two sides of the same coin [11, 13]. There are substantive differences that make TreeTracker and TTJ different. First the constraint satisfaction problem concerns the existence of a non-empty model for a large logical formula. Thus, constraint satisfaction algorithms including TreeTracker stop execution and return TRUE upon identifying what in a relational query would be just one row of the result. In contrast, TTJ produces all tuples in the query output. Second, the TreeTracker algorithm does not make use of hash tables, and is thus structured like a nested loop join rather than a hash join. This is because unlike the study of queries in databases, constraint satisfaction rarely specializes the problem to only equality predicates. Combining these two differences TreeTracker incorporates ad-hoc data structures, where TTJ employs recognized indices commonly used in databases. These difference clearly manifest in the respective complexity analyses. The complexity of the best variation of the TreeTracker algorithm is polynomial in the input size and does not consider the output size. We prove below TTJ runs in linear time in the total size of the input and output.

## 3 Preliminaries

In this section, we present the foundational concepts concerning acyclic join queries and the specific definitions adopted in this paper.

### 3.1 Join Queries and Acyclicity

We consider natural join queries, also known as *full conjunctive queries*, of the form:

$$Q(\boldsymbol{x}) = R_1(\boldsymbol{x}_1) \bowtie R_2(\boldsymbol{x}_2) \bowtie \cdots \bowtie R_n(\boldsymbol{x}_n) \tag{1}$$

where each $R_i$ is a relation name, each $\boldsymbol{x}_i$ (and $\boldsymbol{x}$) a tuple of distinct variables, and every $x \in \boldsymbol{x}_i$ also appears in $\boldsymbol{x}$. We call each $R_i(\boldsymbol{x}_i)$ an *atom*, and $x_i$ the *schema* of $R_i$, denoted as $\Sigma(R_i)$. We extend the notion of schema to tuples in the standard way and write $\Sigma(t)$ for the schema of $t$. The query computes the set[4] $Q = \{\boldsymbol{x} \mid \bigwedge_{i\in[n]} \boldsymbol{x}_i \in R_i\}$. We sometimes write $Q$ and not $Q(\boldsymbol{x})$ to reduce clutter, and identify $Q$ with its set of relations. For example, $Q - \{R_i\}$ denotes the query $Q$ with $R_i$ removed.

*Definition 3.1 (Join Tree).* A *join tree* for a query $Q$ is a tree where each node is an atom in $Q$, such that for every variable $x$, the nodes containing $x$ form a connected subtree.

A query $Q$ is *acyclic* (more specifically $\alpha$-acyclic) if there exists a *join tree* for Q.
For clarity we rewrite the query in example 1.1 and detail one of its join trees:

$$Q_1 = R(i, x) \bowtie S(x, y, j) \bowtie T(y, k) \bowtie U(y, l) \tag{2}$$

One join tree has $R(i, x)$ at the root, $S(x, y, j)$ as its child, and $T(y, k)$ and $U(y, l)$ as children of $S$. We encourage the reader to draw a picture of this join tree for reference. One can construct a join tree for any acyclic query with the GYO algorithm [9, 23], which works by finding a sequence of *ears*. To define ear, we first define a *key schema*:

[4]For clarity we assume set semantics. No change is needed for TTJ to support bag semantics

```
def GYO(Q):
  forest = { tree(R) for R in Q }
  while not Q.is_empty():
    R = find-ear(Q)
    P = parent(Q, R)
    forest.set_parent(R, P)
    Q.remove(R)
  return forest
```

(a) The GYO alglorithm.

```
def parent(R, Q):
  if Q.is_empty(): return None
  keys = Σ(R) ∩ ⋃_{S∈Q-{R}} Σ(S)
  for S in Q - {R}:
    if keys ⊆ Σ(S):
      return S
  # we did not find a valid parent
  return None
```

(b) Find a parent of $R$ in $Q$ if one exists.

Fig. 2. GYO reduction and parent computation.

*Definition 3.2 (Key Schema).* For a query $Q$ of the form (1), the *key schema* of an atom $R_i(\boldsymbol{x}_i)$ in $Q$, denoted as $\text{keys}(Q, R_i)$, is the set of variables shared between $R_i(\boldsymbol{x}_i)$ with the other atoms in $Q$; i.e., $\text{keys}(Q, R_i) = \boldsymbol{x}_i \cap \bigcup_{j \in [n] \wedge j \neq i} \boldsymbol{x}_j$.

Intuitively, $\text{keys}(Q, R_i)$ form the keys of $R_i$'s hash table, if we compute $(Q - \{R_i\}) \bowtie R_i$ using binary hash join.

*Definition 3.3 (Ear).* Given a query $Q$ of the form (1), an atom $R_i(\boldsymbol{x}_i)$ is an *ear* if it satisfies the property $\exists j \neq i : \boldsymbol{x}_j \supseteq \text{keys}(Q, R_i)$. In words, there is another atom $R_j(\boldsymbol{x}_j)$ that contains all the variables in $R_i$'s key schema. We call such an $R_j$ a *parent* of $R_i$.

The parent concept is central to the TTJ algorithm. The parent's schema include all of its children's keys. When a hash lookup fails at a child, TTJ will backjump to the parent. Figure 2b shows an algorithm to find the first parent of an ear in $Q$, where $Q$ is represented as a list of atoms.

The GYO algorithm for constructing join trees is shown in Figure 2a: we start with a forest where each atom makes up its own tree, then for every ear, we attach it to its parent and remove that ear from the query. Note that it is possible for the algorithm to produce a forest of disjoint trees when the query contains Cartesian products. For simplicity, we will ignore such cases.

*Definition 3.4 (GYO reduction order).* Given a query $Q$ of the form (1), a *GYO reduction order* for a query $Q$ is a sequence $[R_{p_1}, R_{p_2}, \ldots, R_{p_n}]$ that is a permutation of $[R_1, R_2, \ldots, R_n]$, such that for every $i < n$, the atom $R_{p_i}$ is an ear in the (sub)query $R_{p_i} \bowtie \cdots \bowtie R_{p_n}$.

Equivalently, it is the same order of atoms as visited by the GYO algorithm. The reader can verify $[U, T, S, R]$is a GYO reduction order for $Q_1$. The existence of a GYO reduction order and the existence of a join tree are equivalent.

Theorem 3.5 ([9, 23]). *A query $Q$ has a join tree (i.e., $Q$ is $\alpha$-acyclic) if and only if it has a GYO reduction order.*

### 3.2 Binary Hash Join

In this paper we focus on hash-based join algorithms. For theoretical analyses we focus on left-deep linear plans; for practical implementation we follow the standard practice and decompose each bushy plan into a sequence of left-deep linear plans, materializing each intermediate result.

*Definition 3.6 (Query Plan).* A (left-deep linear) query plan for a query $Q$ of the form (1) is a sequence $[R_{p_1}, R_{p_2}, \ldots, R_{p_n}]$ that is a permutation of $Q$'s relations $[R_1, R_2, \ldots, R_n]$.

```
1 def join(t, plan, i):
2   if i > plan.len(): print(t)
3   else:
4     R = plan[i]; k = π_{keys(plan[1..i],R)}(t)
5     for r in R[k]:
6       join(t++r, plan, i+1)
```

(a) Pipelined left-deep binary hash join

```
1 def YA(Q, order):
2   for R in order: # semijoins reduction
3     P = parent(Q, R); Q.remove(R)
4     if P is not None: P = P ⋉ R
5   # compute the output with hash join
6   return join((), reverse(order), 1)
```

(b) Yannakakis's algorithm

Fig. 3. Binary hash join and Yannakakis's algorithm. The plan array is 1-indexed.

For consistency we adopt 1-based indexing for query plans, so the first relation in the plan is stored at $i = 1$. An example query plan for $Q_1$ in 2 is $[R, S, T, U]$. One may notice similarities between a GYO reduction order and a query plan. The reason for this will become clear.

We follow the push-based model [14] and specialize the binary hash join algorithm for pipelined left-deep plans as shown in Figure 3a. We write $\pi_s(t)$ to project the tuple $t$ onto the schema $s$, and $t +\!\!+ r$ to concatenate the tuples $t$ and $r$ while resolving the schema appropriately. Execution begins by passing to `join` the empty tuple $t = ()$, a query plan, and $i = 1$. Although we do not need to build a hash table for the left-most relation (the first relation in the plan), for simplicity we assume that there is a (degenerate) hash table mapping the empty tuple $()$ to the entire left-most relation. The algorithm starts by checking if the plan has been exhausted and if so, output the tuple $t$. Otherwise, we retrieve the $i$-th relation $R_{p_i}$ from the plan, and lookup from $R_{p_i}$ the matching tuples that join with $t$. For each match, we concatenate it with $t$ and recursively call `join`.

It may be helpful to unroll the recursion over a query plan, and we encourage the reader to do so for $Q_1$ in (2) with the plan $[R, S, T, U]$. This will generate the same code as in Figure 1a.

### 3.3 Yannakakis's Algorithm

Yannakakis's original algorithm [22] makes two preprocessing passes over the input relations. A third pass computes the joins yielding the final output. Bagan, Durand, and Gandjean [2] improved the original algorithm by eliminating the second preprocessing pass. For brevity we only describe the latter algorithm. Following common usage, hereafter, we will refer to the improved version as Yannakakis's alglorithm (YA).

Shown in Figure 3b is, given a GYO reduction order, the relations are preprocessed using semijoins, then the output is computed with standard hash join. Equivalently, the semijoin preprocessing step can be performed by traversing a join tree bottom-up, and the output computed with hash join by traversing the tree top-down.

*Example 3.7.* Given the query $Q_1$ in (2) and the GYO reduction order $[U, T, S, R]$, YA first performs the series of semijoins, $S' = S \ltimes U$, $S'' = S' \ltimes T$, and $R' = R \ltimes S''$, then computes the output with the plan $[R', S'', T, U]$. The reader may refer to the join tree of $Q_1$ and confirm we are traversing the tree bottom-up then top-down.

## 4 TreeTracker Join

The TreeTracker Join algorithm is shown in Figure 4a. The algorithm follows the same structure as binary hash join. The difference starts on line 5 right before the hash lookup `R[k]`. If this lookup fails (i.e., it finds no match), and if $R$ has a parent $P$ that appears before $R$ in the plan, then TTJ backjumps to the **`for`**-loop at $P$'s recursive level, by returning `P` (line 7). This is similar to throwing an exception which is "caught" at the loop level of `P`, as we will explain on line 10. Otherwise, if the

```
1  def ttj(t, plan, i):
2    if i > plan.len(): print(t)
3    else:
4      R = plan[i]; k = π_{keys(plan[1..i],R)}(t)
5      P = parent(plan[1..i], R)
6      if R[k] is None & P is not None:
7        return P # backjump to P
8      for r in R[k]:
9        result = ttj(t⧺r, plan, i+1)
10       if result == R: # catch backjump
11         R[k].delete(r)
12       elif: result is not None:
13         return result # continue backjump
```

(a) TreeTracker join

```
1  if R[()] is None: throw BJ(None)
2  for i,x in R:
3    try: if S[x] is None: throw BJ(R)
4      for y,j in S[x]:
5        try: if T[y] is None: throw BJ(S)
6          for k in T[y]:
7            try: if U[y] is None: throw BJ(S)
8              for l in U[y]:
9                try: output(x,y,i,j,k,l)
10               catch BJ(U): U[y].delete(l)
11           catch BJ(T): T[y].delete(k)
12       catch BJ(S): S[x].delete(y,j)
13   catch BJ(R): R[()].delete(i, x)
```

(b) Execution of TTJ for Example 1.1

Fig. 4. The TreeTracker algorithm and an example execution.

lookup `R[k]` succeeds, the algorithm iterates over each matching tuple $r$ and calls itself recursively (line 9). This recursive call has three possible results. A result containing a relation (line 10) signifies a backjump has occurred, with that relation as the backjumping point. If the backjumping point is the same as the current relation $R$, then the tuple $r$ is deleted from $R$ (line 11). If the backjumping point is different from $R$, then the backjump continues by returning `result` which interrupts the current loop. Finally, if the recursive call (implicitly) returns `None`, the algorithm continues to the next loop iteration.

*Example 4.1.* It can be helpful to unroll the recursive algorithm over a query plan. Given $Q_1$ in (2) and the plan $[R, S, T, U]$, Figure 4b shows the execution of TTJ. To make the code more intuitive, we replace **`return`** statements with exception handling to simulate backjumping. We gray out dead code and no-ops:

- Line 1 is unreachable because `R[()]` is always the entire relation $R$, and $R$ has no parent.
- Line 3 (and 13) is a no-op, because it would just backjump to the immediately enclosing loop, and removing a tuple from $R$ is useless because $R$ is at the outermost loop[5].
- Technically the if-statement on line 5 is useful even though it only backjumps one level, because the backjump would remove a tuple from $S$ when caught (line 12). However for the input data in Example 1.1 we do not need this, and we gray it out to reduce clutter.
- Finally, the innermost two try-catch pairs are unreachable, because $U$ and $T$ have no children.

At this point, the remaining code in black is essentially the same as the code in Figure 1c. As a side note, a sufficiently smart compiler with partial evaluation or just-in-time compilation could remove the dead code and no-ops as we have done above.

### 4.1 Correctness and Asymptotic Complexity

The correctness proof starts with an observation on the relationship between different calls to `ttj`:

Proposition 4.2. *If* $\texttt{ttj}(t_j, p, j)$ *recursively calls* $\texttt{ttj}(t_i, p, i)$, *then* $t_j \subseteq t_i$.

Proof. The proposition follows from the definition of the algorithm, where the $t_i$ argument to the nested call is constructed by appending tuples to $t_j$. □

[5]In Section 5 we will introduce an additional optimization that makes "removing" from the outermost relation meaningful.

TTJ differs from binary join only upon a lookup failure. In that case it backjumps to the parent of the relation that caused the failure, and deletes the tuple that caused the failure. Therefore, TTJ is correct as long as it never deletes or "backjumps over" any tuple that should be in the output. We first prove that a deleted tuple can never contribute to any outupt. In the following we write $\pi_R(t)$ for the projection of $t$ onto the schema of $R$.

Lemma 4.3. *Suppose a tuple $r_j$ is deleted from $R_j$ during the execution of TTJ for a query $Q$ using plan $p$. Then $\forall t_{out} \in Q : \pi_{R_j}(t_{out}) \neq r_j$.*

Proof. Let $p$ be $[R_1, \ldots, R_n]$, and $t_j$ be the value of the argument $t$ in scope at the time of the deletion. Because $r_j$ is deleted from $R_j$, there must be a failed lookup $R_i[k_i]$ recursively nested within the call to $\texttt{ttj}(t_j +\!\!+ r_j, p, j+1)$, and $R_j$ is the parent of $R_i$. Let $K_i = \text{keys}(p[1, \ldots, i], R_i)$, and let $t_i$ be the value of $t$ at the time of the lookup failure. Then $t_j +\!\!+ r_j \subseteq t_i$ by Proposition 4.2. By definition of parent, $K_i \subseteq \Sigma(R_j) \subseteq \Sigma(t_j +\!\!+ r_j) \subseteq \Sigma(t_i)$, so $k_i = \pi_{K_i}(t_i) = \pi_{K_i}(t_j +\!\!+ r_j) = \pi_{K_i}(r_j)$. However, since the lookup failure implies no tuple in $R_i$ contains $k_i$, any output tuple $t_{\text{out}}$ cannot contain $k_i$ either, i.e., $\forall t_{\text{out}} \in Q : \pi_{K_i}(t_{\text{out}}) \neq k_i$. Therefore, $\forall t_{\text{out}} \in Q : \pi_{K_i}(t_{\text{out}}) \neq \pi_{K_i}(r_j)$ which implies $\forall t_{\text{out}} \in Q : \pi_{R_j}(t_{\text{out}}) \neq \pi_{R_j}(r_j)$. □

Next, we show TTJ never backjumps over any tuple that contributes to the output. Given a plan $p = [R_1, \ldots, R_n]$, denote by $\pi_{[i]}(t)$ the projection of $t$ onto $\bigcup_{j \in [i]} \Sigma(R_j)$.

Lemma 4.4. *For any tuple $t_{out} \in Q$, plan $p$ for $Q$, and $1 \leq i \leq |p|$, $\texttt{ttj}(\pi_{[i-1]}(t_{out}), p, i)$ recursively calls $\texttt{ttj}(\pi_{[i]}(t_{out}), p, i+1)$.*

Proof. Consider a lookup $R[k]$ that is recursively nested within the call to $\texttt{ttj}(\pi_{[i-1]}(t_{\text{out}}), p, i)$ where $R$ has a parent $R_j$ with $j \in [i-1]$. Then $k \subseteq \pi_{R_j}(t_{\text{out}}) \subseteq t_{\text{out}}$, and because $\Sigma(k) \subseteq \Sigma(R)$, we have $k \subseteq \pi_R(t_{\text{out}}) \in R$. This means the lookup $R[k]$ will not fail. This holds for all such $R$, so the algorithm never backjumps from within the call $\texttt{ttj}(\pi_{[i-1]}(t_{\text{out}}), p, i)$ to any $R_j$ for $j \in [i-1]$. The algorithm may still backjump to $R_i$, but by Lemma 4.3, $\pi_{R_i}(t_{\text{out}})$ is never deleted from $R_i$, and therefore the algorithm will recursively call $\texttt{ttj}(\pi_{[i-1]}(t_{\text{out}}) +\!\!+ \pi_{R_i}(t_{\text{out}}), p, i+1)$ which is the same as $\texttt{ttj}(\pi_{[i]}(t_{\text{out}}), p, i+1)$. □

We arrive at the correctness of TTJ by applying Lemma 4.4 inductively over the query plan.

Theorem 4.5. *Given any plan* `p` *for $Q$,* `ttj((),p,1)` *computes $Q$.*

Proof. We prove the correctness of TTJ in two directions: first, any tuple produced by TTJ should be in the output; second, TTJ produces all tuples that should be in the output. The first direction is straightforward, as any tuple produced by TTJ is also produced by binary hash join. We prove the second direction by induction over the argument $i$, with the following inductive hypothesis: $\texttt{ttj}(\pi_{[i-1]}(t_{\text{out}}), p, i)$ will be invoked for all $t_{\text{out}} \in Q$ and $1 \leq i \leq |p|$. The base case when $i = 1$ holds because we start the execution of TTJ by calling $\texttt{ttj}((), p, 1)$. For the inductive step, assume $\texttt{ttj}(\pi_{[i-1]}(t_{\text{out}}), p, i)$ is invoked, then applying Lemma 4.4 shows $\texttt{ttj}(\pi_{[i]}(t_{\text{out}}), p, i+1)$ will also be invoked. Therefore, $\texttt{ttj}(t_{\text{out}}, p, |p|+1)$ will be invoked for all $t_{\text{out}} \in Q$, which produces all tuples that should be in the output. □

Next, we prove TTJ runs in linear time in the size of the input and output, for full acyclic queries. We first introduce a condition on the query plan that is necessary for the linear time complexity:

Lemma 4.6. *Given a query $Q$ and a plan $p = [R_1, \ldots, R_n]$ for $Q$,* `parent` *returns* `None` *only for $R_1$ during the execution of TTJ, if $p$ is the reverse of a GYO reduction order of $Q$.*

Proof. If $[R_n, \ldots, R_1]$ is a GYO reduction order, then there is a join tree with $R_1$ as root, and every non-root atom has a parent. □

TTJ is guaranteed to run in linear time given such a plan:

Theorem 4.7. *Fix a query $Q$ and a plan $p$. If $p$ is the reverse of a GYO reduction order for $Q$, then* `ttj(),p,1)` *computes $Q$ in time $O(|Q| + \sum_i |R_i|)$.*

Proof. We first note that in Figure 4a, `ttj` does constant work outside of the loops; each iteration of the loop also does constant work and recursively calls `ttj`, so each call to `ttj` accounts for constant work, therefore the total run time is linear in the number of calls to `ttj`. All we need to show now is that there are a linear number of calls to `ttj`.

Because $p$ is the reverse of a GYO reduction order for $Q$, the following holds from Lemma 4.6: except for the one call to `ttj` on the root relation (when $i = 1$), every call to `ttj` has 3 possible outcomes: (1) It outputs a tuple. (2) It backjumps and deletes a tuple from an input relation. (3) It recursively calls `ttj`. Because the query plan has constant length, there can be at most a constant number of recursive calls to `ttj` (case 3) until we reach cases 1 or 2. Therefore there are at most $O(|Q| + \sum_i |R_i|)$ calls to `ttj`, and the algorithm runs in that time. □

By Theorem 3.5 every $\alpha$-acyclic query can be GYO-reduced, therefore `ttj` runs in linear time:

Corollary 4.8. *For any $\alpha$-acyclic query $Q$, there is a plan $p$ such that* `ttj(),p,1)` *computes $Q$ in time $O(|Q| + \sum_i |R_i|)$.*

### 4.2 Comparison with Binary Join and YA

We now prove our claim that, for any given query plan, TTJ always matches or outperforms binary hash join. Because TTJ and hash join build the exact same set of hash tables, they share the same cost for hash building. We therefore focus on the cost of hash lookups which accounts for the majority of the remaining cost for both algorithms. The following proofs take advantage of set semantics, but it is easy to extend the reasoning for bag semantics, as we can convert a bag into a set by appending a unique labeled null value to each tuple. We start with the following observation to relate the run time of hash join and TTJ to the set of arguments they are invoked with:

Lemma 4.9. *Both hash join and TTJ, as defined in Figure 3a and Figure 4a, are invoked once for each distinct combination of the arguments $(t, p, i)$.*

Proof. We prove by induction over the argument $i$. In the base case when $i = 1$, both algorithms are invoked once with $t = ()$, $i = 1$. For the inductive step, first consider the hash join algorithm. For every distinct $t$, `join(t, plan, i)` recursively calls `join(t++r, plan, i+1)` for every $r \in R_i[k]$. Since $R_i$ is a set, each $r$ is distinct, so each $t +\!\!+ r$ is also distinct. The same reasoning also applies to TTJ, as the algorithm will call itself only for a subset of the tuples in $R_i[k]$. □

In other words, the number of calls to each algorithm is the same as the number of distinct arguments they are invoked with. We can now compare the algorithms, by bounding the number of calls to TTJ by that of binary join.

Theorem 4.10. *Given a query $Q$ and a plan $p$ for $Q$, computing $Q$ with TTJ using $p$ makes at most as many hash lookups as computing $Q$ with binary join using $p$.*

Proof. For clarity we have repeated the lookup `R[k]` three times in Figure 4a, but we really only need to look up once and save the result to a local variable for reuse. Specifically, a pointer to `R[k]` on line 6 can be used for the nullness check on the same line, the loop on line 8, as well as the deletion[6]on line 11. This way, every call to `ttj` makes exactly one hash lookup. Since the

[6]Although the deletion occurs after a recursive function call, the recursion has constant depth, so the pointer dereference has good temporal locality and is likely cheap.

binary join algorithm in Figure 3a also makes exactly one hash lookup per call, it is sufficient to bound the number of calls to `ttj` by that of binary join. By Lemma 4.9, it is sufficient to show the distinct arguments TTJ is invoked on is a subset of that for binary join. We prove this by induction over the argument $i$. When $i = 1$, both TTJ and binary join are invoked with $t = ()$ and $i = 1$. For the inductive step, `ttj(t, p, i)` recursively calls `ttj(t++r, p, i+1)` only if $r \in R_i[k]$, which implies `join(t, p, i)` will also call `join(t++r, p, i+1)` in binary join. Therefore, every call to TTJ is accounted for with a call to binary join. □

Another cost in query execution comes from accessing the matching tuples after a successful lookup, and one can prove that TTJ accesses no more tuples than binary join, following the same reasoning as above. Although backjumping and tuple deletion in TTJ may in principle carry an overhead, we will show in Section 6 that such an overhead is negligible as compared to the cost of hash lookups. Finally, we note the above proof does not assume an acyclic query. Section 7 analyzes the run time of TTJ on cyclic queries.

While we guarantee TTJ to always match binary join, we cannot make the same strong claim for YA. We will see in Section 6 that YA performs better than TTJ on some queries. Here we analyze a few extreme cases for some intuition of how TTJ compares to YA:

*Example 4.11.* Consider a query where every tuple successfully joins, i.e., no lookup fails. In this case binary join and TTJ behaves identically. However, YA spends additional time futilely computing semijoins (without removing any tuple), before following the same execution as binary join and TTJ to produce the output.

*Example 4.12.* The other extreme case is when a query has no output, and YA immediately detects this and stops. In fact Example 1.1 is such a query: all YA needs to do is the semijoin $T \ltimes U$, where it builds a (tiny) hash table for $U$ and iterate over $T$ once to detect nothing joins. In contrast, although TTJ also runs in linear time, it must build the hash table for all of $S$, $T$ and $U$.

## 5 Optimizations

Up until this section TTJ has been presented in foundational manner, requiring only minor changes to HJ. Deep consideration of TTJ reveals many opportunties for enhancement. We present two direct optimizations of the TTJ algorithm inspired by research in Constraint Satisfaction. We name these the *deletion propagation* and *no-good list* optimizations. Deletion propagation is emodied in the TreeTracker algorithm [10] and we include it to examine its effectiveness on join evaluation. No-good list is also known as *no-good recording*, which stems from the *constraint learning* method in Constraint Satisfaction [7].

*Deletion Propagation.* Recall that after a lookup failure, a backjump is executed and the offending tuple removed it from its relation based on the corresponding hash key. There will be executions where all the tuples sharing that hash key are removed. Programatically in line 11 in Figure 4a `R[k]` becomes empty. If so any subsequent lookup, `R[k]` will fail. Instead of continuing execution, as defined so far, we can immediately backtrack further to the parent of $R$ and *propagate* the deletion to $R$'s parent. Said optimization requires adding the single following line to the end of Figure 4a:

```
if R[k] is None & P is not None: return P
```

This optimization is not always beneficial. When there are no subsequent lookups to `R[k]` propagating the deletion is unnecessary and carries a small overhead.

*No-Good List.* We had remarked in Section 4 that removing a tuple from the root relation is pointless, as the same tuple would never be considered again. However, any tuple in the root relation that shares the same values with an offending tuple over the key schema will also fail. The

no-good list optimization comprises adding that set of values to a blacklist. Each tuple from the root relation is tested for membership in the blacklist. Since membership in that list mean certain failure no further effort to join that tuple is necessary. This optimization requires three changes to Figure 4a.

First, the key values must be included as parameters and passed to the parent relation, line 7:

```
return (P, π_R(t))
```

When catching the backjump (line 10) at the root relation, those key values are added to the blacklist:

```
if result == (R, vals):
  if i == 0: no_good.add(vals) else: R[k].delete(r)
```

When iterating over the root relation, (after line 8), each tuple is tested for membership in the no-good list and if present further processing is skipped.

```
if i == 0 & r.matches(no_good): continue
```

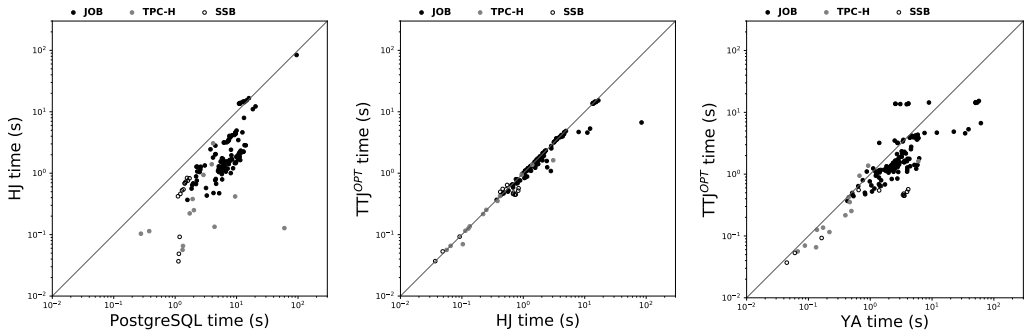
The no-good list, *ng*, can be implemented as a hash table. Suppose the root relation, $R$, has $m$ children $S_1, \ldots, S_m$. The lookup key for *ng* is $\langle S_i, \ell_i \rangle$ where $\ell_i$ is a set containing $\pi_{\text{keys}(R,S_i)}(t)$ (called *no-goods*) for a tuple $t$ from $R$ that caused a lookup failure at $S_i$ for $i \in [m]$. The impact of the no-good list is almost identical to semijoin reduction in YA. The algorithmic difference is in lieu of a semijoin removing dangling tuples prior to the join, the $R$ tuples are checked against a collection of values accumulated on the fly and at anytime during execution are a subset of the contents of the complementary antijoin. Like YA itself, the effectiveness of the no-good list depends on how much the argument is reduced and the size of the intermediate result. i.e. the semijoin and join selectivity. We demonstrate the trade-off through Star Schema Benchmark in Section 6.1.

## 6 Empirical Results

Since our primary contribution concerns the development of an algorithm that is both asymptotically optimal and is competative in practice w.r.t. wall clock time, the primary goal of the empirical assessment is to compare the execution time of the algorithms in as controlled of an experiment as possible. All three algorithms, TTJ, binary hash join, HJ, and YA are implemented in the same Java query execution engine written from scratch. We are certain our algorithm execution measurements do not make calls to methods outside of our execution environment. Any data structure in our execution environment whose definition is impacted by the definition of a data structure outside of our Java execution environment is treated identically for all three algorithms. Where possible, code is reused across algorithm implementation. The source code of the implementation is available at https://anonymous.4open.science/r/treetracker.

Remaining aspects of query compilation and and DBMS implementation are “borrowed” from other DBMS implementations. Query plans are an example of borrowing from other DBMS implementations. After loading a benchmark database instance and gathering catalog statistics left-deep linear query plans are determined by SQLite, and bushy plans by PostgreSQL. The SQL EXPLAIN command elicits the plans from the DBMSs. SQLite and PostgreSQL were chosen because of the topology of the plans their optimizers generate. The linear time guarantee only holds for left-deep linear plans that are consistant with a GYO reduction order of the query. All the left-deep plans produced by SQLite in our experiments are consistent with the GYO reduction requirement.

*Workload.* Our experments encompass left-deep plans, left-deep plans with optimizations integrated into the TTJ algorithm, and bushy plans. Only the acyclic join queries in three benchmarks were evaluated, the Join Ordering Benchmark (JOB) [12], TPC-H [18] (scale factor = 1), and the Star Schema Benchmark (SSB) [16] (scale factor = 1). Also omitted were single-relation queries, and




(a) HJ vs. PostgreSQL (b) TTJ vs. HJ (c) TTJ vs. YA

Fig. 5. Run time of TTJ, HJ, YA, and PostgreSQL on JOB, TPC-H, and SSB. Every data point corresponds to a query, whose $x$- and $y$-coordinates correspond to the run time of the algorithms under comparison.

| Baseline | Benchmark | Average | Maximum | | Mininimum | |
|---|---|---|---|---|---|---|
| Hash Join | JOB | 1.11× | 12.6× | (16b) | 0.9× | (11b) |
| | TPC-H | 1.09× | 1.9× | (Q9) | 1× | (Q7) |
| | SSB | 1.15× | 1.7× | (Q2.2) | 0.8× | (Q3.4) |
| Yannakakis's Algorithm | JOB | 1.60× | 9.2× | (16b) | 0.2× | (6a) |
| | TPC-H | 1.40× | 3.7× | (Q9) | 0.7× | (Q7) |
| | SSB | 3.16× | 7.9× | (Q2.2) | 1× | (Q3.4) |

Table 1. Speed-up of TreeTracker Join Relative to Hash Join and Yannakakis's Algorithm.

correlated subqueries. These criteria eliminated only 9 queries, all from TPC-H. Thus, the 113 JOB queries, the 13 SSB queries and 13 out of 22 TPC-H queries were assessed, for a total of 139 queries.

*Environment.* Experiments were conducted on a single logical core of an AMD Ryzen 9 5900X 12-Core Processor @ 3.7Hz CPU. The computer contained 64 GB of RAM, and a 1TB PCIe NVMe Gen3 M.2 2280 Internal SSD. All data structures are allocated from JVM heap which was set to 20 GB. Since execution was otherwise identical for all algorithms under test, no techniques to reduce the overhead of memory allocation or garbage collection were exploited. Measurements for each query and algorithm pair were orchestrated by JMH [1] configured for 5 warmup forks and 10 measurement forks. Each of those forks contains 3 warmup and 5 measurement iterations.

Direct measurements of PostgreSQL, which can be seen as the control group (not a baseline), for our implementations are the same as [12]. PostgreSQL measurements use an in-memory hash join, indices were dropped and single process execution specified. Thus, we configured PostreSQL such that measurements were made as similar to our Java implementations as we could make possible. A timeout was set to 1 minute. Of all the queries only TPC-H Q20 exceeded the timeout.

### 6.1 Algorithm Comparison

Figure 5 illustrates our primary results. It contains 3 scatter plots that pairwise compare the execution time of 4 implementations for each query across the 3 benchmarks. First, Figure 5a compares the performance of PostgreSQL, using hash joins with our implementation of HJ. Inspection of

the scatterplot shows that with few exceptions the execution time of the same query is less than an order of magnitude apart. Most points are below the diagonal indicating our implementation is faster than PostreSQL. The shape of the cluster suggests a consistent range in the disparity of execution time.

Faster execution is not surprising. The results of a road race with PostgreSQL are not material to this paper. Our execution environment contains no elements of transaction system overhead or buffer and memory hierarchy management. PostgreSQL execution time was measured as a control. This first plot establishes that our Java implementation is within range of a commercially used RDBMS and the consistency in the difference of execution speed lends credibility that the emperical results from our execution environment will generalize to commercially deployed RDBMSs.

Figure 5b shows, on a per query basis, the relative speed of TTJ versus HJ. The visualization in Figure 5b reveals that TTJ is often faster than HJ, and for just a few queries the execution is slower and when that is the case the performance disadvantage is marginal. Per Table 1, JOB query 11b and SSB query Q3.4, form the worst results for TTJ are just 10% and 20% slower respectively. The weighted average of TTJ execution time over the three benchmarks is a hair better than 10% faster. More sizable improvements appear in the maximum speed-up results. We remind the reader the join orders are for plans that were optimized for left-deep linears hash-joins. Below we will return to the question of the upside opportunity for TTJ execution speed when, in future work, a SQL optimizer includes cost models for TTJ and the optimization process includes both a choice of join order and a choice of join algorithm. For instance the detailed examination of each query execution revealed that TTJ's worst-relative performance, JOB query 11b is due to the inclusion of the no-good list optimization which, often predictably, incures overhead without providing any performance benefit.

For completeness, performance of TTJ relative to YA is presented in Figure 5c. The results exemplify the paradox and challenge of YA. On all but 12 queries, TTJ outperforms YA, with average and maximum speedup of 1.4x and 9.2x. 8 of those queries, JOB 6a, 6b, 6c, 6d, 6e, 7b, 12b, and TPC-H Q7, exhibit the most significant disadvantage of TTJ.

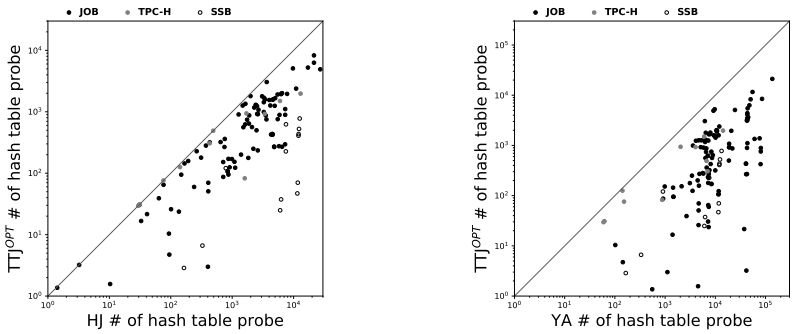
Review of the JOB queries reveals a foreseeable cause for YA execution speed advantage. The first semijoin removes a large fraction of tuples from a large relation. For example, the first semijoin for JOB query 6a reduces the largest relation, cast_info, from 36,000,000 tuples to 486 tuples. That semijoin is executed before building the hash tables. Hash table build time for YA is 499ms. For TTJ that build time is 13,398ms and by itself comprises 98% of the execution time for TTJ.

The basis of TPC-H, Q7's performance results are also due to the impact of the first semijoin, but in a more involved way. Prior to any join processing a relational select on nation returns just 1 tuple. As an argument to the first semijoin, supplier $\ltimes$ nation, over 90% of tuples from supplier are removed. Where, in the first example the one semijoin reduction accounted for speed benefit, in this example, by beginning with a single tuple, the entire chain of semijoin reductions resulted in large reductions in the size of the join arguments repeat.

Review of hash table build times for YA relative to hash table build times for TTJ and HJ alone, (these latter two always being equal), accounts for all the speed improvement of YA compared to the other algorithms.

The specialized pattern embodied in SSB, star queries on a star schema, enables a quantitative assessment that may be used in the future by a query optimizer. Notably a determination if the integration of a no-good list is advantagous. For the special case of big data queries modeled by SSB, the performance of TTJ is largely determined by the effectiveness of the no-good list.

Recall the no-good list is a specialization for the leftmost argument of a join plan as hash-joins do not typically create a hash-table for the leftmost argument. The no-good list forms a cache of the tuple key values for the leftmost argument that have been determined to be dangling. Queries plans




(a) TTJ vs. HJ (b) TTJ vs. YA

Fig. 6. Number of hash probes in different algorithms.

for star schema typically start with the fact table as the leftmost relation in a plan, and provide the key values for a series of joins on dimension tables. Any lookup failure will backjump to the fact table and add to the no-good list. Thus the no-good list acts as a filter that prevents any processing of a fact tuple whose join key values have already been determined to be fruitless.

Of 13 SSB queries evaluated TTJ is the fastest algorithm on 6, and, plus or minus, within 10% margin of the best algorithm on 10 queries. We compared the queries that run relatively slower in TTJ (Q1.2, Q3.4, Q4.1, and Q4.3) with those that run relatively faster (Q2.1, Q2.2, Q2.3, Q3.1, Q3.2, and Q3.3), and measured the ratio between the intermediate result size reduction and the size of no-good lists. We determined that for the slower queries, each element in the no-good list, on average, reduces the intermediate result size by 182. For the faster queries the average is 318. Although an optimizer is not within the scope of this paper, we can conclude that for our testbed more refined measurements would determine a tipping point value of a selectivity that falls between 1/182 and 1/318. Selectivity below the tipping point indicates omitting the no-good list will result in faster query execution and vice versa.

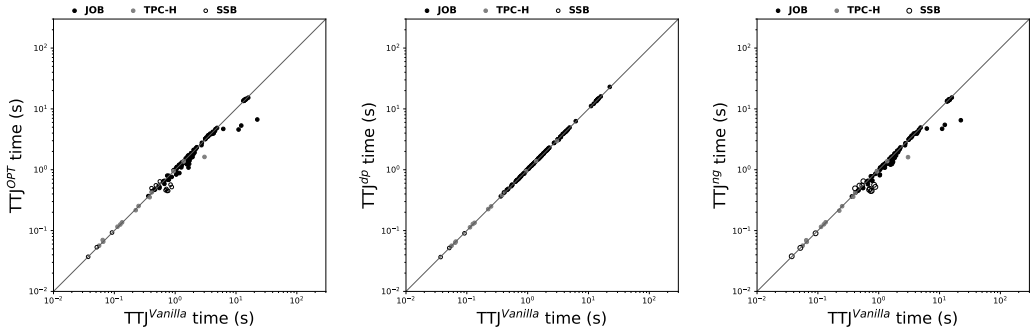
The scatter plot Figure 6a compares the number of hash probes for TTJ vs. HJ for each query. A small number of the scatter plot points appear on the diagonal, i.e. an equal number of hash probes. The remainder of the points are below the diagonal. This emperically validates our theorem that TTJ will execute fewer or an equal number of hash probes as HJ.

We have not made any claims as to the relative number of hash probes between TTJ and YA. Nevertheless we made that measurement. Figure 6b shows like HJ, TTJ makes fewer hash probes for YA. Yet YA runs faster for certain queries as hash building, not probing, sometimes dominates query run time, as we have pointed out in the analysis of results in Figure 5c.

### 6.2 Impact of Optimizations

Experiments in this section investigate the impact of the two optimizations introduced in Section 5, no-good list and deletion propagation. To denote TTJ with both no-good list (*ng*) and deletion propagation (*dp*) we write $\text{TTJ}^{ng+dp}$. To denote TTJ with no-good list only and TTJ with deletion propagation only we write $\text{TTJ}^{ng}$ and $\text{TTJ}^{dp}$ respectively. For this section TTJ shall mean the algorithm without the optimizations. Figure 7 contains scatter plots that compare the runtime of TTJ with each of the three possible integrations of the optimizations.

Reviewing the behavior of the optimizations independently, we see that deletion propagation has little effect on query run time. This is due to the fundamental differences between constraint




(a) Using both optimizations (b) Deletion propagation (*dp*) (c) No-good list (*ng*)

Fig. 7. Performance impact of TTJ optimizations.

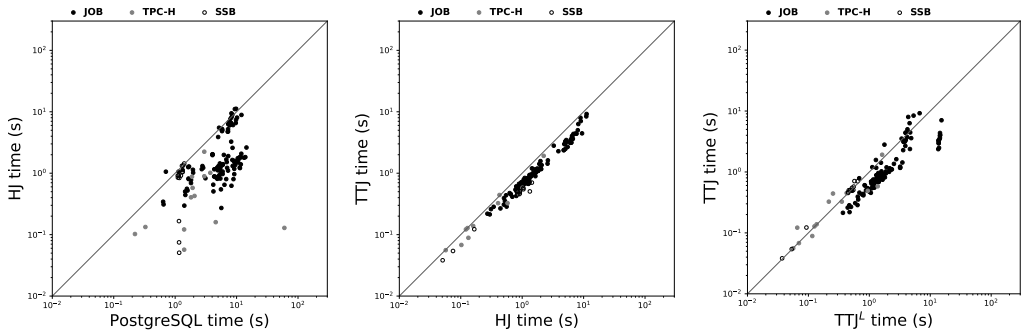
satisfaction and query evaluation. In constraint satisfaction, the *modus operandi* is to backtrack upon a failed constraint, and such failures are common due to the large number of constraints present. In contrast, in query evaluation the number of constraints is small as the query is usually much smaller than the input data, and a large number of tuples will eventually satisfy all constraints and appear in the output. Indeed, from running the entire benchmark suite, only around 5% of deletions trigger propagation.

On the other hand, no-good list markedly improves the performance in a number of queries. There are only a few queries slightly above the diagonal, all from the SSB benchmark. Recall from our SSB analysis (Section 6.1) that the effectiveness of the no-good list hinges on the ratio of intermediate result size reduction to the no-good list size. For star schema queries, this ratio is dominated by the no-good list's ability to filter tuples from the fact table—the left-most relation in the join plan. Quantifying this, we observe that each no-good element reduces intermediate results by 318 on average for fast queries (Q2.1, Q2.2, Q2.3, Q3.1, Q3.2, Q3.3) versus 182 for slow queries (Q1.2, Q3.4, Q4.1, Q4.3). The fast queries achieve a 75% higher reduction per element, surpassing the threshold where benefits outweigh costs. Crucially, this aligns with scenarios where the join plan structure and backjump dynamics prioritize filtering the left-most relation—a pattern common in efficient executions.

### 6.3 TTJ on Bushy Plans

Since every plan in Section 6.1 is compatible with a GYO reduction order, the runtime of TTJ is guaranteed to be linear. However, given the abundance of bushy plans (all the native PostgreSQL plans we used here are bushy), a natural question to ask is whether TTJ can still provide reasonably good performance despite the loss of linear runtime guarantee. The results in this section give an affirmative answer.

Figures 8a and 8b compare the run time of TTJ, HJ, and PostgreSQL using bushy plans produced by PostgreSQL, and Figure 8c compares the run time of TTJ using bushy plans with the same algorithm using left-deep linear plans produced by SQLite, denoted by $\text{TTJ}^L$. Figure 8a shows our HJ baseline remains competitive with PostgreSQL under bushy plans. TTJ is faster than HJ on all 113 JOB queries, and faster than $\text{TTJ}^L$ on 101 (89%) of them. Compared to $\text{TTJ}^L$, the maximum speed-up is 5.7× (7b), the minimum speed-up is 0.5× (19d), and the average speed-up (geometric mean) is 1.75×. Compared to HJ, the maximum speed-up is 2.1× (13d), the minimum speed-up is 1.1× (19d), and the average speed-up (geometric mean) is 1.56×. From the figure, we observe that the




(a) HJ v.s. PostgreSQL (b) TTJ v.s. HJ (c) TTJ v.s. TTJ$^L$

Fig. 8. Run time comparison of TTJ, TTJ$^L$, HJ and PostgreSQL. TTJ$^L$ uses left-deep linear plans generated by SQLite (same as earlier experiments), while all other algorithms use bushy plans generated by PostgreSQL.

materialization of intermediate results due to bushy plans does not degrade TTJ performance; in fact, TTJ performs much better than itself on linear plans in most cases due to the fact that intermediate results generated in bushy plans are usually smaller than some of the largest input relations in JOB, which allows TTJ to spend less time building hash tables. As a result, the saving in join computation becomes more salient than that over left-deep plans. Furthermore, the performance improvement using TTJ on each individual linear plan has the compound effect that contribute to the overall performance improvement of the queries. However, we do observe that some of the queries still have better performance under linear plans than bushy plans such as 8c and 16b. This indicates that linear time guarantee is still meaningful for query performance and optimization is still necessary to bring out the best performance of TTJ (e.g., decide which plan shape to use). An important topic for future work is to optimize bushy plans that also have the guarantee of optimality.

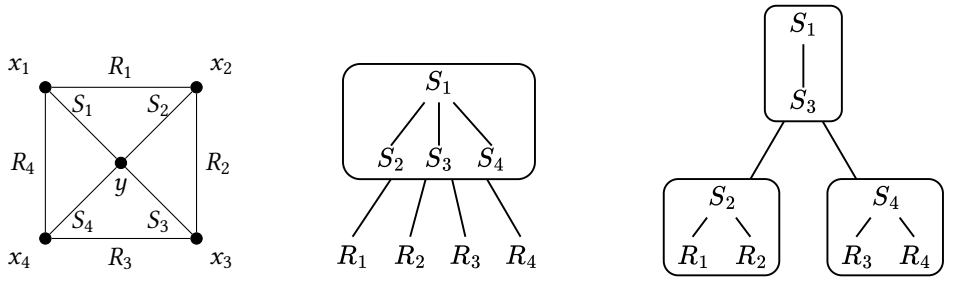
## 7 Cyclic Queries

The TTJ algorithm as defined in Figure 4a supports both acyclic and cyclic queries. The guarantee to match or outperform binary join (Theorem 4.10) also holds for cyclic queries. However, the linear-time guarantee only applies to acyclic queries. To analyze the run time of TTJ on cyclic queries, we introduce a new method called *tree convolution* to break down a cyclic query in to acyclic parts. The next example illustrates the intuition behind tree convolution.

*Example 7.1.* Consider the following query whose query graph is shown in Figure 9a:

$$Q_{\boxtimes} :\text{-} R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_1) \bowtie$$
$$S_1(x_1, y) \bowtie S_2(x_2, y) \bowtie S_3(x_3, y) \bowtie S_4(x_4, y)$$

In the query graph, each node is a variable, and there is an edge between two nodes if the corresponding variables appear in the same atom, for example the edge $R_1$ between $x_1$ and $x_2$ corresponds to the atom $R_1(x_1, x_2)$. Clearly the query is cyclic. However, we can compute the query by breaking it down into two acyclic steps: First, we compute the join $S_1 \bowtie S_2 \bowtie S_3 \bowtie S_4$ and store the result in a temporary relation $S$. Then we compute the final result with $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie S$. Any acyclic join algorithm can be used to compute each step, and, as we will show later, using TTJ can avoid materializing the intermediate result $S$. The total run time is therefore $O(|\text{IN}| + |\text{OUT}| + |S|)$.




(a) Quey graph of $Q_{\boxtimes}$. (b) A tree convolution of $Q_{\boxtimes}$. (c) Anothre tree convolution of $Q_{\boxtimes}$.

Fig. 9. Query graph and tree convolutions of $Q_{\boxtimes}$ in Example 7.1.

To formally define tree convolutions, we first identify a (full conjunctive) query with the set of atoms in its body, and define a *subquery* of a query $Q$ as a subset of the atoms in $Q$. Then, a tree convolution for a query is a nested tree, defined recursively as follows:

*Definition 7.2.* A tree convolution of a query $Q$, written $\mathcal{T}(Q)$, is a tree where each node is either an atom in $Q$, or a tree convolution of a subquery of $Q$. Each atom in $Q$ appears exactly once anywhere in $\mathcal{T}(Q)$, and every tree $T$ in $\mathcal{T}(Q)$ forms a join tree of the corresponding (sub-)query, after replacing each non-atom node $v \in T$ with a fresh atom over all variables in $v$.

Figure 9b shows a tree convolution corresponding to the computation of $Q_{\boxtimes}$ in Example 7.1: first join together the $S$ relations, then join the result with the $R$ relations. Figure 9c shows a different convolution, where we first compute three acyclic subqueries, then join together the results. Another convolution with four nesting levels is shown in Figure 10b.

We can compute any cyclic query using a tree convolution. Starting from the most deeply nested trees, we run any acyclic join algorithm to materialize intermediate results, until we reach the top-level tree that produces the final output. However, TTJ can avoid the expensive materialization by using a special kind of tree convolution called *rooted convolution*.

*Definition 7.3.* A convolution is *rooted* if nested convolutions only appear at the root of each tree.

The convolution in Figure 9b is rooted, while the one in Figure 9c is not. We can generate a plan $p$ for TTJ by traversing a rooted convolution inside-out: starting with the most deeply nested tree, initialize $p$ with the reverse of the GYO-reduction order of this tree; then, as we go up each level, append the reverse of the GYO-reduction order to the end of $p$.

*Example 7.4.* The rooted convolution in Figure 9b generates the plan $[S_1, S_2, S_3, S_4, R_1, R_2, R_3, R_4]$.

A small adjustment to the TTJ algorithm is necessary to fully exploit rooted convolutions. If we execute TTJ as-is using the plan in Example 7.4, none of the relations $R_1, \ldots, R_4$ have a parent in the plan, yet we need to compute the join $S \bowtie R_1 \bowtie \cdots \bowtie R_4$ in time $O(|S| + \sum_i |R_i| + |Q|)$, where $S$ is the join of $S_1, \ldots, S_4$. We therefore introduce additional backjumps from each $R_i$ to $S_4$, but without deleting any tuple from $S_4$. This is achieved by defining the `parent` function to work over rooted convolutions: given a tree convolution $C$ and a relation $R$, if the parent node of $R$ in some tree of $C$ is an atom $P$, then assign $P$ as the parent of $R$; otherwise if the parent node is a nested tree, then assign the last relation in that tree (i.e. the first relation in its GYO-reduction order) as the parent of $R$. For example, the parent of each $R_i$ in Example 7.4 is $S_4$.

Finally, when backjumping to a parent in a nested convolution, we do not delete any tuple from that parent. We are now ready to analyze the run time of TTJ on cyclic queries.

Given a rooted tree convolution $\mathcal{T}(Q)$, we generate a query plan $p$ as follows. Suppose $\mathcal{T}(Q)$ consists of $m$ nested trees $T_1, \ldots, T_m$ where $T_1$ is the most deeply nested tree and $T_m$ is the outermost tree. Let $p_1, \ldots, p_m$ be the plans corresponding to $T_1, \ldots, T_m$. Then, each $p_i$ is a plan that corresponds to the reverse of a GYO-reduction order of $T_i$, where the first relation in $p_i$ is result of $p_{i-1}$ for $i > 1$. The outermost plan $p_m$ then computes the final result of $Q$.

Proposition 7.5. *During* `ttj` *execution on a given rooted convolution $\mathcal{T}$ of a query $Q$, if a lookup fails at $R$ that belongs to $p_i$ but not in $p_{i-1}$ ($i \geq 2$),* `ttj` *either backjumps to an atom that is in $p_i$ but not in $p_{i-1}$ or backjumps to the last atom of $p_{i-1}$.*

Proof. For any $p_i$ with $i \geq 2$, since $p_i$ is the reverse of a GYO-reduction order of the $i$-th tree of $\mathcal{T}$ and by the modified `parent` function, every relation in $p_i$ but not in $p_{i-1}$ has parent. Furthermore, if lookup fails at $R$ that is in $p_i$ but not in $p_{i-1}$, `ttj` backjumps to $R$'s parent. Then, the result follows by the definition of the modified `parent` function. □

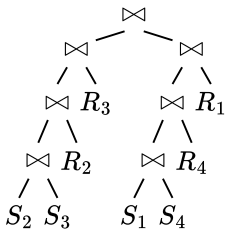
In $p_1$, if the parent of a relation is the first relation of $p_1$, `ttj` backjumps to the first relation. We can treat the first relation of $p_1$ as $p_0$ (i.e., the most deeply nested tree in $\mathcal{T}$ is now a node of an atom of $Q$) and the relation is also the last atom of $p_0$. Therefore, we can remove the restriction of $i \geq 2$ in Proposition 7.5. In the following proof, we reference the Proposition 7.5 with the understanding that it holds for $i \geq 1$.

Theorem 7.6. *Given a rooted convolution $\mathcal{T}(Q)$ of $Q$, there is a plan $p$ such that TTJ runs in time $O(|IN| + |OUT| + \sum_i |S_i|)$ on $p$ where $|S_i|$ is the size of the join of all relations in the $i$-th tree of $\mathcal{T}(Q)$.*

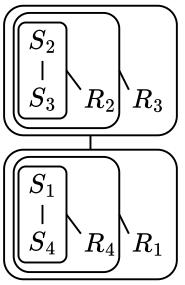
Proof. Since there is no change to `ttj` except using the modified `parent` function for $\mathcal{T}$, like Proof of Theorem 4.7, we only need to show the algorithm makes $O(|\mathsf{IN}| + |\mathsf{OUT}| + \sum_i |S_i|)$ number of calls to `ttj`.

Excep for the initial call to `ttj` with `ttj(), p, 1`), every call to `ttj` has three possible outcomes: (1) It outputs a tuple. (2) It backjumps and possibly deletes a tuple from an input relation. (3) It recursively calls `ttj`. Because the query plan has constant length, there can be at most a constant number of recursive calls to `ttj` (case 3) until we reach cases 1 or 2. There can be $O(|Q|)$ `ttj` calls for case 1. Since lookup cannot fail at the first relation of $p$, the relation that lookup fails at is in some $p_i$ but not in $p_{i-1}$. Let $R$ be a relation that a lookup fails at. By Proposition 7.5, there can be two cases on where `ttj` backjumps to. If `ttj` backjumps to an atom that is also in $p_i$ but not in $p_{i-1}$, a tuple is deleted. This case can happen $O(|\mathsf{IN}|)$ times. If `ttj` backjumps to the last atom of $p_{i-1}$, since `ttj` works no different than binary join from this moment until next lookup failure, this can happen $O(|S_{i-1}|)$ times; the tuples computed at the last atom of $p_{i-1}$ is $S_{i-1}$. Therefore, given $1 \leq i \leq m$, there are at most $O(|\mathsf{IN}| + |\mathsf{OUT}| + \sum_{i=1}^{m-1} |S_i|)$ calls to `ttj`, and the algorithm runs in that time. □

We conclude this section by noting that tree convolution generalizes several exisiting ideas in databases and constraint satisfaction. First, classic binary join plans are a special case, where every tree in the tree convolution is of size 2. For example, the convolution in Figure 10b corresponds to the binary join plan in Figure 10a. In other words, traditional hash join can be thought of as computing a convolution one binary join at a time, and we have generalized this to computing a multi-way acyclic join at a time. In a similar way, rooted convolutions generalize left-deep linear plans, as the top-half of Figure 10b corresponds to the left-half in Figure 10a. Second, in constraint satisfaction a *cycle cut set* is a subset of the constraints whose exclusion makes the constraint problem acyclic. In database terms, it is a subset of the atoms of $Q$ whose removal leaves an acyclic subquery. Tree convolution generalizes cycle cut sets in the sense that it "cuts" the query in multiple rounds, with each round producing acyclic subqueries that can be computed in linear time. Finally,




(a) A binary join plan for $Q_{\boxtimes}$.




(b) A tree convolution for $Q_{\boxtimes}$.

Fig. 10. A binary join plan for the query in Example 7.1 and the corresponding tree convolution.

many of the properties of tree convolution are shared with tree decomposition of hypergraphs [8] in database theory. In fact both tree convolutions in Figure 9 can also be seen as tree decompositions: Figure 9b is a tree decomposition with 5 bags, one containing $\{S_1, S_2, S_3, S_4\}$, and one for each $R_i$; each box in Figure 9c forms a bag in the corresponding tree decomposition. These are then also examples of where algorithms using tree decompositions require materializing the join result of each bag. But given a rooted tree convolution, Figure 9b, TTJ requires no materialization.

## 8 Future Work and Conclusion

In this paper we have proposed our new join algorithm, TreeTracker Join (TTJ). The algorithm runs in time $O(|\text{IN}| + |\text{OUT}|)$ on acyclic queries, and guarantees to make no more hash probes than binary hash join on the same query plan. We have shown empirically that TTJ is competitive with binary hash join and Yannakakis's algorithm.

Although our implementation already beats PostgreSQL in our experiments, challenges remain for TTJ to compete with highly optimized systems. Decades of research on binary join has produced effective techniques like column-oriented storage, vectorized execution, and parallel execution, just to name a few. Future research should investigate how to adapt these techniques to TTJ.

Another avenue for future work is to develop a dedicated query optimizer for TTJ. As this paper's focus is on algorithm-level comparison, we have opted to reuse existing systems to produce binary hash join plans, which are then executed using TTJ. Tailoring the optimizer to TTJ may yield plans with better performance. For instance, estimating the number of hash probe failures instead of intermediate result sizes shall more accurately model the execution cost of TTJ. On the other hand, extending TTJ to also guarantee a linear time complexity on bushy plans is also an interesting challenge. Our theoretical analyses of TTJ reveal a close connection between GYO-reduction orders and left-deep linear plans, both of which are total orders. Since bushy plans and join trees both define partial orders, we conjecture there exists a algorithm that runs in linear time on any bushy plan that corresponds to a join tree, with the same guarantee of matching binary hash join on the same plan.

Finally, our experiments focus on acyclic queries due to their prevalence in traditional workloads. However, with the rise of graph databases practitioners begin to encounter more and more cyclic queries. Additional research on TTJ for cyclic queries, both in terms of practical performance and theoretical guarantees, will be very valuable. Some open problems include: Given any hypergraph, what is the minimum nesting depth of any tree convolution? How does this number related to other measures like various notions of hypergraph widths? And what is the complexity of

finding the optimal tree convolution given a cost function? Answering theses questions will aid the development of a query optimizer for TTJ on cyclic queries.

## References


[1] [n. d.]. Java Microbenchmark Harness (JMH). https://github.com/openjdk/jmh

[2] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Annual Conference for Computer Science Logic*. https://api.semanticscholar.org/CorpusID:15398587

[3] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4646)*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, 208–222. doi:10.1007/978-3-540-74915-8_18

[4] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2024. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. *CoRR* abs/2411.04042 (2024). doi:10.48550/ARXIV.2411.04042 arXiv:2411.04042

[5] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (2024), 3215–3228. doi:10.14778/3681954.3681995

[6] Rina Dechter. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artif. Intell.* 41, 3 (1990), 273–312. doi:10.1016/0004-3702(90)90046-3

[7] Rina Dechter. 2003. *Constraint Processing*. Morgan Kaufmann, USA.

[8] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 57–74. doi:10.1145/2902251.2902309

[9] M. Graham. 1980. *On the universal relation*. Technical Report. University of Toronto, Computer Systems Research Group.

[10] Roberto J. Bayardo Jr. and Daniel P. Miranker. 1994. An Optimal Backtrack Algorithm for Tree-Structured Constraint Satisfaction problems. *Artif. Intell.* 71, 1 (1994), 159–181. doi:10.1016/0004-3702(94)90064-7

[11] Phokion G. Kolaitis and Moshe Y. Vardi. 2000. Conjunctive-Query Containment and Constraint Satisfaction. *J. Comput. Syst. Sci.* 61, 2 (2000), 302–332. doi:10.1006/JCSS.2000.1713

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

[13] Daniel P. Miranker, Roberto J. Bayardo, and Vasilis Samoladas. 1997. Query Evaluation as Constraint Search; An Overview of Early Results. In *International Symposium on the Applications of Constraint Databases*. https://api.semanticscholar.org/CorpusID:8644835

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

[15] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. doi:10.1145/3180143

[16] Pat O'Neil, Betty O'Neil, and Xuedong Chen. 2009. *Star Schema Bechmark - Revision 3, June 5, 2009*. resreport. UMass/Boston. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[17] SQLite Documentation. 2024. Query Planning and Optimization. https://www.sqlite.org/optoverview.html#hash_joins. Accessed: 2024-07-24.

[18] Transaction Processing Performance Council (TPC). [n. d.]. TPC-H Benchmark. Online. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf Accessed on 11-18-2021.

[19] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2022. Any-k Algorithms for Enumerating Ranked Answers to Conjunctive Queries. *CoRR* abs/2205.05649 (2022). doi:10.48550/ARXIV.2205.05649 arXiv:2205.05649

[20] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2024. Ranked Enumeration for Database Queries. *SIGMOD Rec.* 53, 3 (2024), 6–19. doi:10.1145/3703922.3703924

[21] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR* abs/1210.0481 (2012). arXiv:1210.0481 http://arxiv.org/abs/1210.0481

[22] 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.

[23] Clement T. Yu and M. Z. Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *Annual International Computer Software and Applications Conference*. https://api.semanticscholar.org/CorpusID:7812638

[24] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *Proc. VLDB Endow.* 10, 8 (2017), 889–900. doi:10.14778/3090163.3090167