# TreeTracker Join: Simple, Optimal, Fast

Zeyuan Hu
University of Texas at Austin
Austin, Texas, USA

Remy Wang
University of California, Los Angeles
Los Angeles, California, USA

Daniel P. Miranker
University of Texas at Austin
Austin, Texas, USA

## ABSTRACT

Inspired by the TreeTracker algorithm used in Constraint Satisfaction we present a novel linear-time join algorithm, TreeTracker Join (TTJ). TTJ is very similar to a standard binary hash join, but introduces a test that identifies when a tuple is dangling and removes that tuple from its relation. The test is to simply observe if a hash lookup fails to return any matching tuples. If so, TTJ determines which tuple is responsible for the failure, backtracks to the offending tuple, and removes it from its relation.

As compared to the best known linear-time join algorithm, Yannakakis's algorithm, TTJ shares the same asymptotic complexity on acyclic queries while imposing much lower overhead in practice. We can also reuse any binary join plan for TTJ, with the guarantee that TTJ will match or outperform binary join on the same plan. Furthermore, this guarantee also extends to cyclic queries. Our experiments show TTJ is the fastest algorithm in 97 out of 113 queries, and outperforms binary join and Yannakakis's algorithm by up to 26.7× and 8.9×, respectively.

## 1 INTRODUCTION

In 1981, Yannakakis [11] was the first to describe a linear-time join algorithm (hereafter YA) running in time $O(|\text{IN}| + |\text{OUT}|)$, where |IN| is the input size and |OUT| is the output size. In principle, this is the best asymptotic complexity that 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 makes two passes over the input relations, using semijoins to reduce the size of each input. The reduced relations are then joined to produce the final output. 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 [1] 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 *join tree* which is different from the query plan used by binary joins. Where there is a wealth of techniques to optimize query plans for binary joins, little is known about optimizing join trees for YA.

In this paper, we propose a new linear-time join algorithm called TreeTracker Join (TTJ). Inspired by the TreeTracker algorithm [5] 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 can be determined by the query compiler, as they depend only on the query and not the data. With those in place, TTJ requires no query-execution time preprocessing, and the algorithm's performance is guaranteed to match or outperform binary hash join given the same query plan (Section 3.2).

Thanks to the straightforward nature of TTJ, we are able to craft pleasantly simple proofs of its correctness and efficiency. We use the following example to illustrate 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 $\{i, x\}$. Let the relations be defined as follows:

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

In the above we denote the set $\{1, \ldots, N\}$ with $[N]$. Note that on these input relations, the join produces no output, because $U$ shares no common $y$-values with $S$ or $T$. Let us first compute the join with binary hash join. Suppose the optimizer produces a left-deep join plan $((R \bowtie S) \bowtie T) \bowtie U$. Following this plan, the execution engine first 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 $U$. Then we compute the join as shown in Figure 1a[1]. For each $(i, x)$ tuple in $R$, we probe into the hash table for $S$ to get the $(y, j)$ values. For each $(y, j)$, we probe into $T$, and for each $k$ probe into $U$. Although the query produces no output, the execution here takes $\Omega(N^3)$ time because it essentially 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 on line 7, the algorithm continues to the next iteration of the loop over $k$ values (line 6), even though it will use the same $y$ to probe into $U$ again! To address this, **the first key idea of TTJ is to backjump[2] to the level causing the probe failure.** To keep the presentation simple, we abuse exception handling to implement backjumping as shown in Figure 1b. Upon a failed probe into $U$, we throw an exception which is caught at the end of the second loop level, because the lookup key $y$ was introduced at that level. We then continue to retrieve the next $y, j$ values, skipping over unnecessary iterations over $k$ 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 $S$. We can 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, we remove the current tuple $(x, y, j)$ from $S$. This is safe to do, because we know the $y$ value will always fail to join with $U$. In this way, we remove all tuples from $S$ 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.

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, TTJ is guaranteed to

[1] One may also recognize this as indexed nested loop join, which is equivalent [10].
[2] 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 [5] for a precise definition.

```
# S: x -> [(y, j)]
# T: y -> [k]
# U: y -> [l]
for i,x in R:
  for y,j in S[x]:
    for k in T[y]:
      for l in U[y]:
        print(x,y,i,j,k,l)
```

**(a) Binary join execution**

```
for i,x in R:
  for y,j in S[x]:
    try: for k in T[y]:
      if U[y] is None: throw Backjump
      for l in U[y]:
        print(...)
    catch Backjump:
      # continue to the 2nd loop level
```

**(b) Backjumping**

```
for i,x in R:
  for y,j in S[x]:
    try: for k in T[y]:
      if U[y] is None: throw Backjump
      for l in U[y]:
        print(...)
    catch Backjump:
      S[x].delete((y, j))
```

**(c) Tuple deletion**

**Figure 1: Execution of binary hash join, with backjumping, and with tuple deletion. Differences are underlined.**

```
# t: current partial tuple
# plan: query plan
# i: position in plan
def join(t, plan, i):
  if i == plan.len():
    print(t)
  else:
    R = plan[i]
    for m in R[π_R(t)]:
      join(t++m, plan, i+1)
```

**(a) Binary join**

```
def ttj(t, plan, i):
  if i == plan.len():
    print(t)
  else:
    R = plan[i]; P = parent(i, plan)
    if R[π_R(t)] is None & P is not None:
      throw BackJump(P)
    for m in R[π_R(t)]:
      try: ttj(t++m, plan, i+1)
      catch BackJump(R): R[π_R(t)].delete(m)
```

**(b) TreeTracker join**

```
def parent(i, plan):
  if i == 0: return None
  # the keys are the common attributes
  # between R and previous relations
  keys = R.schema ∩ ⋃_{0≤j<i} plan[j].schema
  for S in plan[0..i]:
    if keys ⊆ S.schema:
      return S
  # we did not find a valid parent
  return None
```

**(c) Computing the backjumping point**

**Figure 2: Binary hash join, TreeTracker join, and the `parent` function for computing the backjumping point in TTJ. $\pi_R(t)$ projects the tuple $t$ onto the common schema of $R$ and $t$, and $t$ ++ $m$ appends $m$ to $t$ while resolving their schema appropriately.**

match the performance of binary join[3], even for cyclic queries. 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 little work. In summary, our contributions are:

- 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.
- Improve the performance of TTJ with further optimizations.
- Conduct experiments to evaluate the efficiency of TTJ.

## 2 PRELIMINARIES

This section introduces fundamental concepts concerning acyclic join queries and the associated definitions adopted in this paper.

### 2.1 Join Queries and Acyclicity

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

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

where each $R_i$ is a relation name, and each $\boldsymbol{x}_i$ (and $\boldsymbol{x}$) a tuple of variables. We call each $R_i(\boldsymbol{x}_i)$ an *atom*. The query computes the set[4] $Q = \{\boldsymbol{x} \mid \bigwedge_{i\in[k]} \boldsymbol{x}_i \in R_i\}$. We will omit $\boldsymbol{x}$ and write $Q = \cdots$ to reduce clutter.

We say a query $Q$ is *acyclic* (more specifically $\alpha$-acyclic) if there exists a *join tree* for Q, defined as follows.

*Definition 2.1 (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.

Consider the formal query used as example 1.1 :

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

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, and verify it satisfies the definition above. One can construct a join tree for any acyclic query with the GYO algorithm [3, 12], which works by finding a sequence of *ears*. To define ear, we first introduce the concept of *key schema*:

*Definition 2.2 (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 $Q$; that is, $\text{keys}(Q, R_i) = \boldsymbol{x}_i \cap \bigcup_{j\neq i} \boldsymbol{x}_j$.

Intuitively, $\text{keys}(Q, R_i)$ form the keys of $R_i$'s hash table, if we first join the other relations in $Q$, and then join the result with $R_i$.

[3]We model the cost of execution by number of hash probes and accessed tuples.

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

```
# input: a forest where each tree has one atom
def GYO(Q, forest):
  while not Q.is_empty():
    R = find-ear(Q); P = parent(Q, R)
    forest.set_parent(R, P)
    Q.remove(R)
```

**Figure 3: The GYO alglorithm**

*Definition 2.3 (Ear).* Given a query $Q$ of the form (1), an atom $R_i(\boldsymbol{x}_i)$ is an *ear* if it satisfies the property $\exists p \neq i : \boldsymbol{x}_p \supseteq \text{keys}(Q, R_i)$. In words, there is another atom $R_p(\boldsymbol{x}_p)$ that contains all the variables in $R_i$'s key schema. We call such an $R_p$ 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.

The GYO algorithm is shown in Figure 3: 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.

*Definition 2.4 (GYO reduction order).* A *GYO reduction order* for a query $Q$ is a sequence $[R_{p_1}, R_{p_2}, \ldots, R_{p_k}]$ that is a permutation of $[R_1, R_2, \ldots, R_k]$, such that for every $i < k$, the atom $R_{p_i}$ is an ear in the (sub)query $R_{p_i} \bowtie \cdots \bowtie R_{p_k}$.

Equivalently, it is the same order of atoms as visited by the GYO algorithm. For example, a GYO reduction order for $Q_1$ is $[U, T, S, R]$, as the reader can verify. The existence of a GYO reduction order implies the exitence of a join tree and vice versa:

Theorem 2.5. *A query $Q$ has a join tree (i.e., $Q$ is $\alpha$-acyclic) if and only if it has a GYO reduction order [3, 12].*

## 2.2 Binary Join

In this paper we focus on hash-based join algorithms. Furthermore we consider only left-deep linear joins, as the common approach to handle bushy joins is to decompose into a sequence of left-deep linear joins and materialize each intermediate result.

*Definition 2.6 (Query Plan).* A (left-deep linear) query plan for a query $Q$ of the form (1) is a sequence $[R_{p_1}, R_{p_2}, \ldots, R_{p_k}]$ that is a permutation of $Q$'s relations $[R_1, R_2, \ldots, R_k]$.

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 later.

We follow the push-based model [9] and present the binary hash join algorithm as in Figure 2a: we start by passing to `join` the empty tuple $t = ()$, a query plan, and $i = 0$ to start at the beginning of the plan. 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. In the body of `join`, we first check if the plan has been exhausted and if so, we output the tuple $t$. Otherwise, we retrieve the $i$-th relation $R_i$ from the plan, and lookup from $R_i$ the matching tuples that join with $t$. For each match, we concatenate it with $t$ and recursively call `join`.

```
# input: a GYO reduction order
def YA(Q, order):
  for R in order: # preprocess with semijoins
    P = parent(Q, R); Q.remove(R)
    if P is not None: P = P ⋉ R
  # compute the output with standard hash join
  return join((), reverse(order), 0)
```

**Figure 4: Yannakakis's algorithm**

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 should result in exactly the same code as in Figure 1a.

## 2.3 Yannakakis's Algorithm

Yannakakis's original algorithm [11] makes two preprocessing passes over the input relations. A third pass computes the joins yielding the final output. Bagan, Durand, and Gandjean [1] 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.

As shown in Figure 4, given a GYO reduction order we first preprocess the relations with semijoins, then compute the output with standard hash join. Equivalently, we can also preprocess by traversing a join tree bottom-up while performing semijoins, and compute the output by traversing the tree top-down with hash join.

*Example 2.7.* Given the query $Q_1$ in (2) and the GYO reduction order $[U, T, S, R]$, YA first performs the series of semijoins $S \ltimes U$, $S \ltimes T$, and $R \ltimes S$, then computes the output with the plan $[R, S, T, U]$. The reader may refer to the join tree of $Q_1$ and see that we are indeed traversing the tree bottom-up then top-down.

## 2.4 The TreeTracker Algorithm

As the name suggests, TreeTracker Join is a direct decendent of the TreeTracker algorithm [5] from Constraint Satisfaction. The TreeTracker CSP algorithm resolved Dechter's conjecture [2] that there existed an optimal algorithm for acyclic CSPs free of any preprocessing. This connection between query answering and constraint satisfaction should not be surprising, as it has been noted the problems are two sides of the same coin [6, 8]. While we will not describe the TreeTracker algorithm in full, we highlight the key differences between TreeTracker and TTJ. First, in the context of constraint satisfaction, the original TreeTracker algorithm stops after producing the first satisfying assignment, while TTJ produces all tuples in the query output. Second, the original TreeTracker algorithm does not make use of hash tables, and is more similar to nested loop join as opposed to hash join. Finally, the best known complexity of the original TreeTracker algorithm is polynomial in the input size and does not consider the output size, while we prove TTJ to run in linear time in the total size of the input and output.

# 3 TREETRACKER JOIN

Before explaining the algorithm, we first introduce the helper function `parent` in Figure 2c for determining the backjumping points.

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

**Figure 5: Execution of TTJ for Example 1.1**

Given a position $i$ and a query plan $[R_{p_1}, \ldots, R_{p_k}]$, `parent` returns the first parent of $R_{p_i}$ in the subquery $Q_i = R_{p_1} \bowtie \cdots \bowtie R_{p_i}$, if $R_{p_i}$ is an ear of $Q_i$. Otherwise, it returns `None`.

*Example 3.1.* Consider again $Q_1$ in Example 1.1 and the query plan $[R, S, T, U]$. Calling `parent(i, [R,S,T,U])` with $i \in \{0, 1, 2, 3\}$ returns `None`, `R`, `S`, and `S`, respectively. This is consistent with the join tree we constructed for $Q_1$: $R$ is the root and therefore has no parent, the parent of $S$ is $R$, and the parent of $T$ and $U$ is $S$.

Note that although `parent` ties closely to the concept of join trees, TTJ continues to work even if there is no join tree and `parent` returns `None` more than once.

We are now ready to present the TTJ algorithm in Figure 2b. The algorithm follows the same structure as binary hash join, and the difference starts at the hash lookup `R[`$\pi_R$`(t)]` on line 6. If this lookup fails (i.e., it finds no match) and if $R$ has a parent $P$, TTJ backjumps to the end of the loop at $P$'s level by throwing an exception (line 7). This will be caught by the corresponding **`catch`** block at $P$'s level, upon which we delete the tuple causing the failure from $P$, and continue onto the next iteration at that loop level.

*Example 3.2.* It can be helpful to unroll the recursion of TTJ over a query plan. Given $Q_1$ in (2) and the plan $[R, S, T, U]$, Figure 5 shows the execution of TTJ. We gray out dead code and no-ops:

- Line 1 is unreachable because `R[()]` is always the entire relation $R$, and $R$ does not have a 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 particular 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 neither $U$ nor $T$ has children.

At this point, the remaining code in black is identical to the code in Figure 1c (after replacing `Backjump(S)` with `Backjump`). As a side note, a sufficiently smart compiler with partial evaluation or just-in-time compilation could potentially remove the dead code and no-ops as we have done above.

[5]In Section 4 we will introduce an additional optimization that makes "removing" from the outermost relation meaningful.

## 3.1 Correctness and Asymptotic Complexity

Thanks to its close resemblance to binary join, we can prove TTJ correct by relying on the correctness of binary join:

Theorem 3.3. *Given any plan* `p` *for* $Q$, `ttj(),p,0)` *computes* $Q$.

Proof. Because we know binary join correctly computes $Q$, we only need to prove the different behavior between TTJ and binary join does not affect the output. Specifically, when a hash lookup `R[`$\pi_R$`(t)]` fails, we show it is safe to backjump to $R$'s parent, $P$, and delete $\pi_P(t)$ from $P$. Upon the lookup failure, we know the hash keys $\pi_R(t)$ do not appear in $R$, therefore they also cannot appear in any output tuple of $Q$. By definition, the schema of the parent, $P$, contains all the key attributes of $R$, so $\pi_P(t)$ will not contribute to any output either. It is therefore safe to backjump over any recursive calls with $t \supseteq \pi_P(t)$, and also delete $\pi_P(t)$ from $P$. □

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 3.4. *Given a query* $Q$ *and a plan* $p = [R_{p_1}, \ldots, R_{p_k}]$ *for* $Q$, `parent` *returns* `None` *only for* $R_{p_1}$ *during the execution of TTJ, if* $p$ *is the reverse of a GYO reduction order of* $Q$.

Proof. If $[R_{p_k}, \ldots, R_{p_1}]$ is a GYO reduction order, then there is a join tree with $R_{p_1}$ as root, and every other atom has a parent. □

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

Theorem 3.5. *Fix a query* $Q$ *and a plan* $p$. *If* $p$ *is the reverse of a GYO reduction order for* $Q$, *then* `ttj(),p,0)` *computes* $Q$ *in time* $O(|Q| + \sum_i |R_i|)$.

Proof. We first note that in Figure 2b, `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, and so 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 3.4: except for the one call to `ttj` on the root relation (when $i = 0$), 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 entire algorithm runs in that time. □

By Theorem 2.5 every $\alpha$-acyclic query has a GYO reduction order, therefore `ttj` runs in linear time:

Corollary 3.6. *For any* $\alpha$*-acyclic query* $Q$, *there is a plan* $p$ *such that* `ttj(),p,0)` *computes* $Q$ *in time* $O(|Q| + \sum_i |R_i|)$.

## 3.2 Comparison with Binary Join and YA

We now prove our claim that, for any given query plan, TTJ always matches or outperforms binary join.

Theorem 3.7. *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[`$\pi_R$`(t)]` three times in Figure 2b, but we really only need to look up once and save the result to a local variable for reuse. This way, every call to `ttj` makes exactly one hash lookup. Since the binary join algorithm in Figure 2a also makes exactly one hash lookup per call, it is sufficient to bound the number of calls to `ttj` by that of binary join. Recall that the execution of TTJ differs from binary join only when a lookup fails, upon which TTJ backtracks at least one recursive level and potentially more, while binary join always returns to the immediately enclosing level. Therefore, the number of calls to `ttj` is at most the number of calls to `join` in binary join. □

Intuitively, after a lookup failure binary join may repeat the same lookup again, as we have seen in Example 1.1, while TTJ avoids that by backjumping to the tuple causing the failure and getting a new one. Also note that in the above proof we did not mention tuple deletion – indeed, tuple deletion is only necessary for the linear time complexity. As another cost in query execution comes from accessing the matching tuples after a successful lookup, one can prove that TTJ accesses no more tuples than binary join, following the same argument as above. Finally, we note that the above proof does not assume an acyclic query.

While we guarantee TTJ to always match binary join, unfortunately we cannot make the same strong claim for YA. We will see in Section 5 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 3.8.* 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 3.9.* 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 $U$.

# 4 OPTIMIZATIONS

We now present two optimizations of TTJ to further improve its performance, namely *deletion propagation* and *no-good list*.

*Deletion Propagation.* Recall that after a lookup failure, we backjump to the offending tuple and remove it from its relation. In certain cases, we remove all tuples sharing the same hash key: `R[`$\pi_R$`(t)]` becomes empty after line 10 in Figure 2b. In this case, we know that in any subsequent lookup, `R[`$\pi_R$`(t)]` will fail. Instead of waiting for and wasting the lookup failure, we immediately backjump to the parent of $R$ and *propagate* the deletion to the parent. To implement this, we add the following line to the end of Figure 2b:

```
if R[πR(t)] is None & P is not None: BackJump(P)
```

There is a case where this optimization is not beneficial. When there are no subsequent lookups to `R[`$\pi_R$`(t)]` propagating the deletion is unnecessary and carries a small overhead.

*No-Good List.* We had remarked in Section 3 that removing a tuple from the root relation is pointless, as the same tuple would never be considered again. However, it can be beneficial to "remove" a tuple more programmatically: the idea of the no-good list optimization is to keep a blacklist of attribute values, so that we immediately skip any tuple matching those attributes. Concretely, we change Figure 2b in three places. First, we include the key values when backjumping on line 7:

```
throw BackJump(P, πR(t))
```

Then when catching the backjump (line 10) at the root relation, we add those key values to the blacklist:

```
catch BackJump(R, keys):
  if i == 0: no_good.add(keys) else: R[πR(t)].delete(m)
```

Finally, we skip over any tuple matching the no-good list while iterating over the root relation (after line 8):

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

Like deletion propagation, the benefit of a no-good list is data dependent. When the list gets big, maintaining and probing it may become more expensive than the lookups saved.

# 5 EXPERIMENTS

We compare the speed of TTJ with binary hash join (HJ) and YA on 113 acyclic queries from the Join Ordering Benchmark [7]. We implement all three algorithms in Java, and provide each algorithm with the same query plan produced by SQLite[6]. Every plan we encoutered can be reversed into a GYO order. Thus, YA and TTJ (by Theorem 3.5) are guaranteed to run in linear time. More detail on the implementation and experiments appears in the long version of this paper[4].

Figure 6 shows the speed-up of YA, HJ, and TTJ relative to native SQLite execution. Of 113 JOB queries, TTJ is the fastest algorithm on 97 (86%) of them. Compared to HJ, the maximum speed-up is 26.7× (16b), the minimum speed-up is 1× (1a), and the average speed-up (geometric mean) is 1.30×. Compared to YA, the maximum speed-up is 8.9× (16b), the minimum speed-up is 0.3× (6a), and the average speed-up is 1.67×.

Whenever HJ outperforms TTJ their difference is negligible. YA visibly outperforms TTJ and HJ on queries 6a, 6b, 6c, 6d, 6e, 7b, and 17a. This is a consequence of the semijoin reduction on the largest relation, `cast_info`, which removes a large fraction of tuples before building its hash table. Since both TTJ and HJ build all their hash tables before computing the join, the time to build the larger hash table dominates. An example is query 6a. The semijoin reduction

[6]For q7b the plan generated by MySQL was used. The SQLite plan contains Cartesian products which we do not yet support.

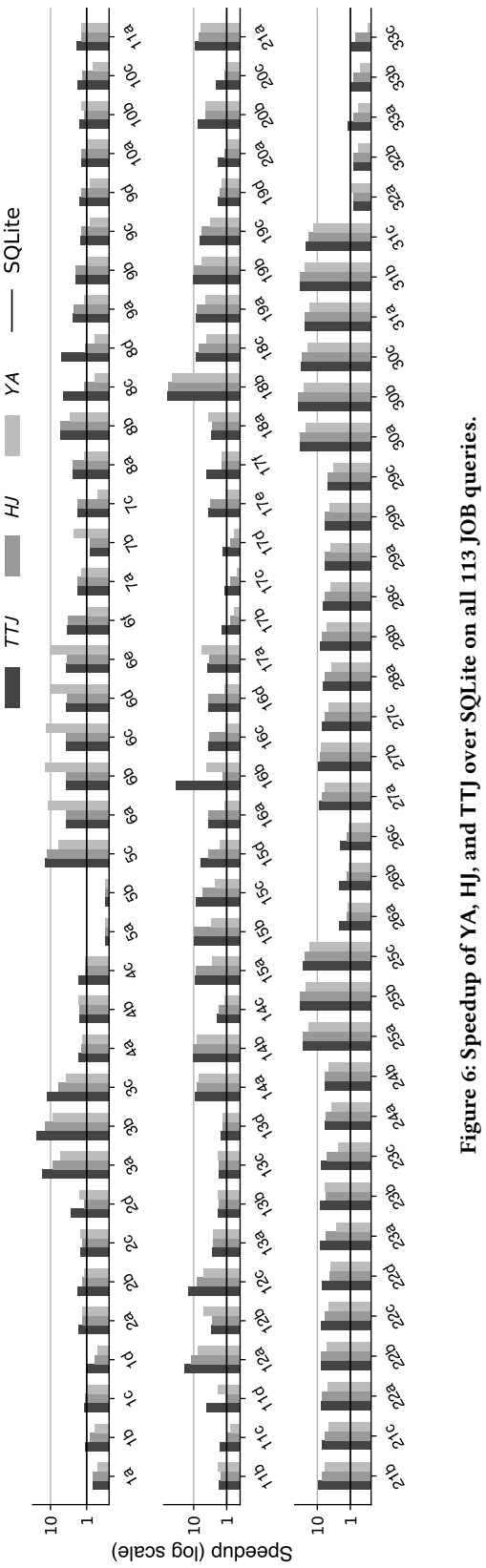


**Figure 6: Speedup of YA, HJ, and TTJ over SQLite on all 113 JOB queries.**

on cast_info reduces around 36 million tuples to 486 tuples. Examining the time taken to build the hash tables for query 6a, YA required 521 ms, which is 8% of the total execution time, compared to 19,710 ms, amounting to 99% of the total runtime for TTJ.

SQLite runs significantly faster on queries 5a and 5b as a result of an optimization we did not implement. These queries return no results. Once SQLite detects the output will be empty it avoids building additional hash tables[7].

## 6 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 given the same query plan it always matches or outperforms binary join. We have shown emperically that TTJ is competitive with binary join and Yannakakis's algorithm.

Although our implementation already beats SQLite 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, or develop optimizations tailored to TTJ like the ones described in Section 4.

Our experiments focued on acyclic queries due to their prevalence in traditional workloads. However, with the rise of graph databases we 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.

## REFERENCES


[1] 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

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

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

[4] Zeyuan Hu and Daniel P. Miranker. 2024. TreeTracker Join: Turning the Tide When a Tuple Fails to Join. *CoRR* abs/2403.01631 (2024). https://doi.org/10.48550/ARXIV.2403.01631 arXiv:2403.01631

[5] 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. https://doi.org/10.1016/0004-3702(94)90064-7

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

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

[8] 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

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

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

[11] 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.


[7] SQLite uses B-trees instead of hash tables, but its documentation [10] treats them as roughly equivalent. For brevity, we refer to SQLite's B-trees as hash tables.

[12] 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