TreeTracker Join: Killing Two Birds With One Stone

September 2, 2023

Zeyuan Hu
University of Texas at Austin
Austin, TX, USA
zeyuan.zack.hu@gmail.com

Daniel P. Miranker
University of Texas at Austin
Austin, TX, USA
miranker@cs.utexas.edu

Bailu Ding*
Microsoft Research
Redmond, WA, USA
bailuding@gmail.com

ABSTRACT

Many important query processing methods introduce semijoin operators or semijoin-like filters to delete dangling tuples, i.e., tuples that do not appear in or impact the final query result. These methods often substantially increase the speed of query execution. However, the application of these methods must be done with care or the cost of the additional operations will outweigh the benefit.

TreeTracker Join (TTJ) is an extension of a hash join operator implemented in iterator interface that detects dangling tuples as it computes join results and integrates their deletion with the execution of a query. The result is a single physical operator that simultaneously computes two logical operations: the join of two relations and a semijoin reduction. For both semijoin and filtering methods, trade-off exists between the cost of the additional operation and potential speed improvement. Thus, many existing methods resort to heuristics specific for each query. TTJ eliminates the use of heuristics and additional operations to attain good performance both formally and empirically. Formally, we prove that computing a k-way acyclic conjunctive query by composing $k-1$ instances of TTJ in a left-deep plan is optimal in data complexity. Empirical results on TPC-H and JOB benchmarks show that TTJ improves query performance on a majority of queries, up to 53%, when compared to hash join. For the remainder the decline in performance is almost always minimal.

PVLDB Reference Format:
Zeyuan Hu, Daniel P. Miranker, and Bailu Ding. TreeTracker Join: Killing Two Birds With One Stone. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Reducing intermediate result size has been shown in practice to correlate with improved join query execution speed [15, 22, 30, 57]. Removing *dangling tuples*, tuples that do not contribute to the final output of a query [18], is a well known approach to achieving that goal. Two usual approaches to identify and delete dangling tuples are: semijoin methods [11, 12, 47, 54–56] and filter methods [15–17, 22, 25, 26, 28, 33, 37, 42, 44, 57]. Semijoin methods like Yannakakis's algorithm and full reducer introduce a sequence of semijoins as a preprocessing step before join evaluation [11, 54]. Such methods may increase plan search space [47] and be prone to cost estimation error [21, 50]. On the other hand, filter methods remove dangling tuples by proactively checking base relations against filters such as bitmap or Bloom filters before computing joins. These methods use filters to create a compact representation of a base relation, for example R_1, and propagate the result downward along the plan to another base relation R_2, where R_1 eventually joins with R_2. If a dangling tuple from R_2 exists and is identified at R_1, such dangling tuple is implicitly removed as part of the filtering process as it fails to pass the check against the filter. The cost of such check, however, can outweigh the benefit of intermediate result size reduction if R_2 is highly selective, in which case existing methods commonly resort to heuristics approaches such as disabling the filters based on selectivity estimation of the underlying relations [15, 44]. Therefore, the query systems using those filter methods are required to assess the trade-off between the execution cost and the potential speed improvement.

TreeTracker Join (TTJ) extends a well known hash join operator (HJ) (Table 1 in [22]) to remove dangling tuples during query execution without introducing additional operators or filter-like data structures into the plan. The key idea is that instead of proactively removing dangling tuples before a join starts, TTJ deletes dangling tuples when they are encountered during query evaluation. Specifically, TTJ assumes all tuples from every relation in the plan constitute the final join result and starts plan evaluation immediately without additional semijoin or filter creation. Once join fails at a TTJ operator, the operator directly resets the evaluation flow to the TTJ operator that causes the join failure, which removes a dangling tuple from the associated hash table. Such reset-and-remove combination potentially skips additional evaluation of multiple operators and avoids generating extra intermediate results including the dangling tuple. HJ was chosen as the starting point because deleting a dangling tuple from a base relation can be implemented by removing the tuple from the hash table built from the relation during the join evaluation. After executing a query using TTJ, the contents of the hash tables closely approximate the result of a semijoin reduction on the relations initially loaded into the hash tables. Thereby, we prove that an ensemble of $k-1$ instances of TTJ in a left-deep plan computes a k-way acyclic join query in $\mathcal{O}(n+r)$ data complexity (the complexity model used in the contemporary formal study of join algorithms [4, 29, 49]) with input size n and output size r. This equals the lower bound for evaluating acyclic conjunctive queries, which makes TTJ optimal.

*Work done outside of Microsoft.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

The TTJ operator is detailed in object-oriented form. The definition of TTJ is consistent with RDBMSs implemented using an iterator-based abstract interface inherited by all query operators [18, 22, 24, 39, 45]. Thus, the addition of TTJ to the existing library of physical operators requires minimal changes to the implementation of those operators.

The use of TTJ does not come without limitations. We only prove the optimality of TTJ on acyclic full conjunctive queries with left-deep query plans where the join operators precede all the cross-products. TTJ can be used in bushy plans that satisfy the same restriction on operation order. A formal construction, tree decomposition [19, 23], when combined with worst-case optimal join algorithms [34, 52], enables the application of algorithms for acyclic CQs, including TTJ, to be used to evaluate cyclic CQs as the last step of the process. Optimality of the operator when used in a bushy plan remains an open issue.

The empirical portion of the paper compares TTJ with HJ on both TPC-H [48] and JOB [30] benchmarks. Compared to HJ, measurements show that TTJ improves individual query performance by up to 53% on a majority of queries. When TTJ performance is not superior, the regression is small, e.g., only 2 out of 126 queries regress by more than 10%.

Organization. We first use a simple example to illustrate how to extend nested-loop join to compute semijoin reduction and join at the same time (§ 2). After preliminaries (§ 3), we detail the join order assumption for left-deep plans to guarantee correctness and optimality of TTJ (§ 4). We present the complete TTJ operator and prove its correctness in § 5. Subsequently, we prove the optimality of TTJ in § 6. § 7 concerns two important issues of using TTJ in practice: cost model and bushy plan. § 8 presents empirical results. We conclude the paper with future work in § 9.

2 MOTIVATING EXAMPLE

The design of TTJ is based on a simple observation illustrated in the example below. TTJ extends the observation to form an operator that can evaluate k-way acyclic join queries.

Example 1. Algorithm 2.1 shows how a simple nested-loop join can be augmented to identify dangling tuples on the fly and compute both the join result and a semijoin result.

Consider a binary join, $T(x)\bowtie S(x)$ evaluated by Algorithm 2.1. Initialized on Line 1, the variable *result* is used to accumulate concatenated pairs of joining tuples. The variable *ng*, short for *no-good list*, is used to accumulate dangling tuples from the relation T. In the outer loop, each tuple, t of T, is checked against a list of tuples of T accumulated in *ng* that have already been determined to not join with any tuples in S (Line 4). If t is known to not join with any $s \in S$, the inner loop is skipped. Execution then moves on to the next iteration of the outer loop. On the other hand, if t is not a member of *ng*, execution continues as expected for a simple nested-loop join. A flag variable, *dangle*, tracks if t fails to join with any tuple in S. If so, t is a dangling tuple and is added to *ng* (Line 10).

Upon termination the join result is returned (Line 12). At this point, the variable *ng* contains all the dangling tuples of T, $T \overline{\ltimes} S$. $T \ltimes S$ is then computed at Line 11. Thus, Algorithm 2.1 computes both $T(x)\bowtie S(x,y,z)$ and $T \ltimes S$. However, the semijoin result is not returned. T can be semijoin reduced if the algorithm assigns T' to T and T is passed into the algorithm as a reference [35].

Algorithm 2.1: Modified Nested-Loop Join to compute join and semijoin at the same time.

```
Input: two relations $T(x)$ and $S(x,y,z)$
Output: join result result and $T' = T \ltimes S$
1  $result, ng \leftarrow \emptyset, \emptyset$
2  for $t \in T$ do
3      $dangle \leftarrow true$
4      if $t \notin ng$ then
5          for $s \in S$ do
6              if $t$ and $s$ are joinable then
7                  add $t \bowtie s$ to $result$
8                  $dangle \leftarrow false$
9          if $dangle = true$ then
10             $ng \leftarrow ng \cup \{t\}$
11 $T' \leftarrow T - ng$
12 return $result$
```

3 PRELIMINARIES

Conjuctive queries (CQs) correspond to select-project-join queries in relational algebra [20]. We focus on *full* CQs, which correspond to a natural join of k relations. A CQ Q is *acyclic* if it admits a join tree $\mathcal{T}_Q$ [9]. A *join tree*, $\mathcal{T}_Q$, is a tree where the nodes represent relations, and it satisfies one additional constraint: for each pair of distinct nodes R, S in the tree and for every common attribute a between R and S, every relation on the path between R and S contains a [9]. A *rooted join tree* is a join tree with one of the nodes chosen to be the root, i.e., converting the tree into a directed tree. For the rest of the paper, we assume $\mathcal{T}_Q$ being a rooted join tree.

Physical operators in an executable query plan are commonly implemented as *iterators* where the consumer of the result of the physical operator can obtain the result one tuple at a time [22]. Iterator, as an interface (in object-oriented context), consists of three methods: open(), getNext(), and close(). open() sets up resources (e.g., necessary data structures) for the computation of the operator; getNext() performs the computation and returns the next tuple in the result; and close() cleans up the used resources. The iterator interface described is shown in Figure 1 (with one more method due to TTJ).

```java
public interface Operator {
  void  open();
  Tuple getNext();
  void  close();
  Tuple deleteDT(Relation R);
}
```

Figure 1: Augmented operator iterator interface illustrated in Java syntax. deleteDT() **is the newly-added method to the interface; the rest is unchanged.**

A *left-deep query plan* $\mathcal{P}_Q$ [39] is a binary tree where internal nodes [14, 40] are join operators $\bowtie$. The right child of any join operator, R_{inner}, is a leaf representing a base relation, which is associated with a table scan operator. The left child of a join operator is a join operator denoted by R_{outer}. TTJ works using demand-driven pipelining, which keeps calling getNext() method of the root join operator of $\mathcal{P}_Q$ until no more tuples are returned [45]. TTJ modifies in-memory hash join (Table 1 in [22]) and thus, we follow the definition from [39] that when used with hash join, $\mathcal{P}_Q$ is evaluated by first creating hash tables associated with each R_{inner} in open() before calling getNext(). Other literature [18, 30] considers the same evaluation strategy with the right-deep query plan, which should not be confused with ours.

We call a relation *internal* if it appears as an internal node [14, 40] in the join tree, $\mathcal{T}_Q$. For relations corresponding to non-root internal nodes of $\mathcal{T}_Q$, we call them *internal*$^{\circ}$ *relation*. Similarly, a *leaf relation* means the relation appears as a leaf node in $\mathcal{T}_Q$. The *root relation* is defined accordingly. Join operators in a plan are labeled top-down as $\bowtie_u$ for $u \in [k-1]$ in ascending order where $[k-1]$ is a shorthand for $1, \ldots, k-1$ [27]. W.l.o.g., we assume relations are labeled top-down in the same fashion as join operators R_1 through R_k[1]. In the later proof, the table scan operator associated with the left-most relation R_k is called $\bowtie_k$. Figure 2 illustrates the notation described. $\mathcal{P}_{\bowtie_u}$ shall denote the set of relations in $\mathcal{P}_Q$ that are below $\bowtie_u$, i.e., $\{R_k, \ldots, R_u\}$. *sort* is a function that extracts attributes from a relation (or from each relation in a set of relations and returns union of the extracted attributes) [3]. In addition, J_u, $u \in [k]$ (including aforementioned $\bowtie_k$), denotes the join result computed by $\bowtie_u$. J_u^* denotes the join of relations in $\mathcal{P}_{\bowtie_u}$. Thus, $\mathcal{P}_Q$ computes the correct join result if and only if $J_1 = J_1^*$. Depending on context we adopt following language usage. A *join fails at* $\bowtie_i$, a *join failure happens at* $\bowtie_i$, or a *join fails at* R_i if a tuple produced from $\bowtie_{i+1}$, the R_{outer} of $\bowtie_i$, cannot join with any tuples from R_i, the R_{inner} of $\bowtie_i$. In such case, R_i is called *join failure relation*. Let j_u denote $\bowtie_u$'s result size. In particular, $j_1 = r$, which is the query result size. We write R^* to denote an R that is free of dangling tuples with respect to Q. For a tuple t of $R(a, b)$, we use an unnamed perspective (e.g., $R(1,\ 2)$) to represent t [3]. $t[a] = \pi_a(t)$ for tuple t, attribute a, and projection π. For tuple t and relations R and S, let *join-attribute value* $jav(t, R, S) = t[sort(R) \cap sort(S)]$. $ja(R, S) = sort(R) \cap sort(S)$. For both HJ and TTJ, $\mathcal{H}_R$ denotes the hash table built from R and $\mathcal{H}_i$ denotes the hash table associated with $\bowtie_i$. *MatchingTuples* denotes the list of tuples that share the same hash key. $R \overline{\ltimes} S = R - (R \ltimes S)$. δ is the distinct operator in relational algebra. Given TTJ as a physical join operator, we use $\mathbb{R}$ to emphasize the physical aspects of a relation R, i.e., a bag of tuples it contains.

We assume a standard RAM complexity model [5]. Following the convention of research in the formal study of join algorithms [4, 29, 49], we use data complexity (in big-$\mathcal{O}$ notation) as the measure of TTJ theoretical performance, which assumes that the size of a query, k, is fixed, but data size n varies [6]. We determine TTJ performance in combined complexity [51] (big-O notation) as well. Combined complexity includes k. Under data complexity, the lower bound of any join algorithm is $\Omega(n + r)$ [49] because the algorithm has to read input relations and produce join output. A join algorithm is *optimal* if its performance upper bound matches the aforementioned lower bound.

[1]We can always rename relations to conform to this notation.

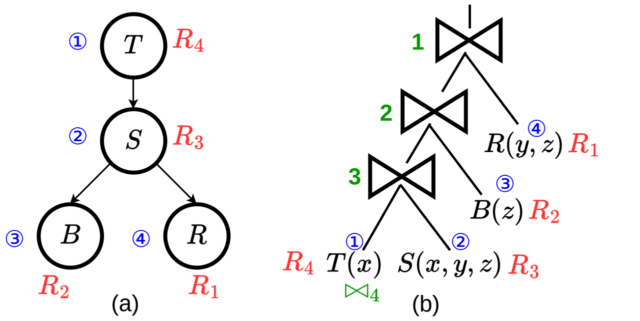

Figure 2: (a) join tree $\mathcal{T}_Q$ and (b) query plan $\mathcal{P}_Q$ for a query Q. ①, ②, ③, ④ show the join order from $\mathcal{T}_Q$ to $\mathcal{P}_Q$. $R_1, \ldots, R_4$ show the relation numbering and $1, 2, 3, \bowtie_4$ denote the join operator numbering in our notation. In specific, $\bowtie_4$ represents the table scan operator associated with R_4, which is T in this example.

4 JOIN ORDER ASSUMPTION

Join order in $\mathcal{P}_Q$ is critical for TTJ correctness. Unless noted otherwise, TTJ assumes the following join order in $\mathcal{P}_Q$.

*Definition 1 (**default join order assumption**).* TTJ assumes for a given relation R_i in $\mathcal{P}_Q$, its parent R_j in $\mathcal{T}_Q$ is to the left of R_i, i.e., $j > i$. Thus, R_k is the root of $\mathcal{T}_Q$.

Effectively, the default join order corresponds to a top-down pass over $\mathcal{T}_Q$. In other words, Definition 1 maps $\mathcal{T}_Q$ into $\mathcal{P}_Q$, from one rooted tree to another rooted tree, and specifies what mapping is compatible with TTJ, i.e., TTJ operates correctly and optimally for any join order satisfying Definition 1. In practice, some order can be better than the other. Thus, we use a cost model to decide (§ 7.1).

Definition 1 might look restrictive but as shown in the following Lemma 1, TTJ join order assumption encapsulates a broad range of join orders used in practice.

Lemma 1. *Any left-deep plan without cross-product for acyclic queries satisfies the default join order assumption (Definition 1).*

Proof. For a left-deep plan without cross-product for an acyclic CQ $ord = [S_k, S_{k-1}, \ldots, S_1]$, our proof proceeds by showing ord permits a rooted join tree that satisfies Definition 1. That is, S_k is R_k, the root of some $\mathcal{T}_Q$, and for any relation S_i, its parent in $\mathcal{T}_Q$ is S_j with $j \in \{k, k-1, \ldots, i+1\}$. For a relation S_i, let attribute set $as(S_i)$ denote the set of attributes appear before it in ord, i.e., $as(S_i) = sort(S_k) \cup \cdots \cup sort(S_{i+1})$. ord has the property that $sort(S_i) \cap as(S_i) \neq \emptyset$. We want to show there is some relation S_j with $j \in \{k, k-1, \ldots, i+1\}$ such that $sort(S_j) \supseteq (sort(S_i) \cap as(S_i))$. If the statement is true, we can construct $\mathcal{T}_Q$ by adding edge (S_j, S_i). To prove the statement, for a relation S_u, suppose relations before S_u already form a join tree, i.e., we are about to attach S_u to the tree. Suppose the statement is not true and there are two more relations S_i, S_j ($i > j > u$) in ord such that $sort(S_u) \cap as(S_u) = (sort(S_i) \cup sort(S_j))$ and $sort(S_u) \cap as(S_u) \not\subseteq sort(S_i)$ (correspondingly for $sort(S_j)$ as well). S_i and S_j are connected via a path. To satisfy join tree requirement, one must add two edges (S_i, S_u) and (S_j, S_u), which form a cycle. □

Lemma 1 enables us to construct $\mathcal{T}_Q$ while deciding order for TTJ using the standard DP approach [18] with TTJ cost model (§ 8): At least one partially-built $\mathcal{T}_Q$ exists for each non-cross-product cell of the DP matrix. Lemma 1 can be generalized to allow cross-products as long as cross-products are padded at the end of the order, i.e., after all the joins.

5 TREETRACKER JOIN OPERATORS

The definition of TTJ in pseudocode, Algorithm 5.1, implements the operator interface shown in Figure 1. TTJ augments the interface with one more method $\mathrm{deleteDT(R)}$. R in the method is a reference [35] to a relation. From now on, we omit the argument R of $\mathrm{deleteDT(R)}$ shown in the figure when refer to the method generically. Methods in TTJ are associated with an operator instance; a k-way join implemented only with TTJ requires $k-1$ instantiations of the operator in $\mathcal{P}_Q$. Following the Java definition [36], instance variables, e.g., r_{outer} in Algorithm 5.1, are accessible by all methods of the TTJ operator, which can change the state of those variables during the runtime. For example, through $\mathrm{deleteDT}()$ calls, TTJ operator j manipulates states of TTJ operator i by asking i to remove its tuple (Line 24 [2]). *nil* in our algorithms represents no such value, not the empty set, $\emptyset$. For a given join failure, a TTJ join operator $\bowtie_i$ is the *detection operator* if the join failure happens at $\bowtie_i$ and thus, Line 20 is called. Correspondingly, a TTJ operator is the *removal operator* if the operator removes a dangling tuple (Line 24 or Algorithm 5.2 Line 10) due to the $\mathrm{deleteDT}()$ call from the detection operator.

Compared to traditional join operator where there are two inputs and one output, join operator $\bowtie_i$ from TTJ has three inputs and two outputs: it produces one additional output via $\mathrm{deleteDT}$ $()$ once join fails and it takes in an additional input from another join operator's $\mathrm{deleteDT}()$ call if $\bowtie_i$ is on the path between the detection operator and the removal operator for a join failure.

TTJ relies on Definition 1 because $\mathrm{deleteDT}()$ is *initiated*[3], calling Line 20, each time when a join fails at a relation. Since $\mathrm{deleteD}$ $\mathrm{T}()$ always sends the relation reference downward, the parent of the relation has to sit at the lower part of the plan than the relation. In other words, if $\mathrm{deleteDT}()$ is initiated from $\bowtie_i$ that has R_i as its inner relation, the destination of $\mathrm{deleteDT}()$, where the recursion call of itself stops, is the operator associated with R_i's parent in $\mathcal{T}_Q$ (cf. Algorithmic Element 3).

Remarks on Algorithm 5.2. To attain the optimality, Algorithm 5.2 is used to replace the table scan operator associated with R_k, i.e., any non-R_k relation is associated with a normal table scan operator. Algorithm 5.2 can be viewed as a degenerate TTJ join operator, which takes in one relation instead of two relations as input (cf. Algorithmic Element 4).

To avoid ambiguity, we define P_Q to be a $\mathcal{P}_Q$ satisfying the default join order assumption and all of its join operators using TTJ join operator (Algorithm 5.1) and the table scan operator associated with R_k is Algorithm 5.2.

Algorithm 5.1: TTJ Join Operator

```
   Purpose: An iterator returns, one at a time, the join result
            of $R_{outer}$ and $R_{inner}$.
   Output: A tuple $t \in R_{outer} \bowtie R_{inner}$
 1 class TTJOperator implements Operator
       Instance variables: $r_{outer}$, $R_{inner}$, $R_{outer}$,
                           $MatchingTuples$
 2     void open()
 3         $r_{outer}$, $MatchingTuples$ all initialized to $nil$
 4         $R_{inner}$.open()
 5         Build hash table, $H$, containing all tuples from
             $R_{inner}$. For each tuple, $r_{inner}$, insert it into the
             hash table using the join attribute value(s),
             $jav(r_{inner})$ as the key
 6         $R_{outer}$.open()
 7     Tuple getNext()
 8         if $MatchingTuples \neq nil \wedge MatchingTuples \neq \emptyset$ then
 9             if ($aMatchingTuple \leftarrow MatchingTuples$.next()
                 ) $\neq nil$ then
10                 return the join of $r_{outer}$ and
                     $aMatchingTuple$
11             $r_{outer} \leftarrow R_{outer}$.getNext()
12             if $r_{outer} = nil$ then return $nil$
13         if $r_{outer} = nil$ then $r_{outer} \leftarrow R_{outer}$.getNext()
14         while $r_{outer} \neq nil$ do
15             $MatchingTuples \leftarrow H$.get($jav(r_{outer})$)
16             if $MatchingTuples \neq nil$ then
17                 $aMatchingTuple \leftarrow MatchingTuples$.next()
18                 return the join of $r_{outer}$ and
                     $aMatchingTuple$
19             else
20                 $r_{outer} \leftarrow R_{outer}$.deleteDT($R_{inner}$)
21         return $nil$
22     Tuple deleteDT($Relation\ R$)
23         if $R_{inner}$ is the parent of $R$ in $\mathcal{T}_Q$ then
24             Remove the last returned tuple from
                 $MatchingTuples$
25         else
26             $MatchingTuples \leftarrow nil$
27             $r_{outer} \leftarrow R_{outer}$.deleteDT($R$)
28             if $r_{outer} = nil$ then return $nil$
29         return getNext()
```

5.1 Examples

We present two examples to illustrate TTJ and highlight the key algorithmic elements in the operator design.

5.1.1 Chain Query. We extend the binary join in Example 1 with one more relation $B(z)$. The database instance is $T(red)$, $S(red, 1, 3)$, $S(red, 3, 2)$, and $B(2)$. Suppose $\mathcal{T}_Q$ of the query is $T \rightarrow S \rightarrow B$,

[2]For the rest of the paper, unless noted otherwise, we reference line numbers from Algorithm 5.1.
[3]$\mathrm{deleteDT}()$ can be further triggered on Line 27 but those calls are triggered by Line 20.

Algorithm 5.2: TTJ Table Scan Operator for R_k

```
   Purpose: Table scan operator for $R_k$ that returns tuples
            not in $ng$.
1  class TTJRkTableScan implements Operator
       Instance variable: $ng$
2      void open()
3          $ng \leftarrow \emptyset$
4      Tuple getNext()
5          while has next tuple $t$ do
6              if $jav(t, R_k, R_i) \notin ng$ for all children $R_i$ of $R_k$ in
                 $\mathcal{T}_Q$ then
7                  return $t$
8          return nil
9      Tuple deleteDT(Relation $R$)
10         Put $jav$ extracted from the last returned tuple in $ng$
11         return getNext()
```

which is a degenerate tree, same as P_Q. Figure 3 is an annotated query plan for the pipeline evaluation of the query. Note, numbered from top to bottom, the annotation includes the hash tables and their contents for $\bowtie_1$ and $\bowtie_2$. The hash key is *jav*. The *jav* for $\bowtie_1$ comprises relation B's z value(s). The *jav* for $\bowtie_2$ are the x values of S.

After recursive open() calls, both $\mathcal{H}_B$ and $\mathcal{H}_S$ are populated. Plan evaluation starts with recursive getNext() calls to itself until reaching Line 4 Algorithm 5.2. Since ng is empty, $T(red)$ is returned (Line 7 Algorithm 5.2). r_{outer} is set to $T(red)$ in $\bowtie_2$ (Line 13). Then, $\bowtie_2$ performs a lookup on $\mathcal{H}_S$ (Line 15). The resulting *MatchingTuples* is $[(red, 1, 3), (red, 3, 2)]$. $T(red)$ joins with the first tuple in the list, $S(red, 1, 3)$. The result $(red, 1, 3)$ is immediately piped to $\bowtie_1$ (Line 18).

Algorithmic Element 1. *Instead of checking ng (Algorithm 2.1 Line 4),* TTJ *join operator (Algorithm 5.1) assumes every tuple in the existing hash table is joinable with the given r_{outer}.*

$(red, 1, 3)$ is detected dangling immediately by looking up the *jav* $z = 3$ into $\mathcal{H}_B$ (Line 15). Thus, join fails at B. Since *MatchingTuples* is *nil* and $R_{outer} = \bowtie_2$ in $\bowtie_1$, $\bowtie_1$ calls $\bowtie_2$.deleteDT(B) on Line 20. In $\bowtie_2$'s deleteDT(B), since S (value of R_{inner} in $\bowtie_2$) is the parent of B in $\mathcal{T}_Q$ (Line 23) and $S(red, 1, 3)$ is the last returned tuple, $S(red, 1, 3)$ is dangling and removed from $\mathcal{H}_S$ (Line 24). We call $\bowtie_1$ detection operator because its r_{outer} contains a dangling tuple. We call $\bowtie_2$ removal operator because the detected dangling tuple is removed from its hash table.

Algorithmic Element 2. *Instead of explicitly maintaining ng similar to Algorithm 2.1 and checking against ng for each r_{outer} value, all deleted tuples from a hash table associated with $\bowtie_i$ $(i \in [k-1])$ implicitly form a ng associated with the operator. Such implicit ng is never checked per Algorithmic Element 1.*

Algorithmic Element 3. *The detection operator (e.g., $\bowtie_1$) where a tuple is detected as dangling (e.g., $S(red, 1, 3)$) is different from the removal operator where the said tuple is removed (e.g., $\bowtie_2$). R_{inner} of the detection operator and R_{inner} of the removal operator follows a child-parent relation in $\mathcal{T}_Q$. Such behavior is analogous to backjumping in constraint satisfaction problem (CSP) [41].*

Observe that because both $\mathcal{T}_Q$ and P_Q are degenerate trees, by the default join order, if the detection operator is $\bowtie_i$, the removal operator is $\bowtie_{i+1}$, which appears immediately before the detection operator in P_Q. As a result, deleteDT() in Algorithm 5.1 can be simplified by removing the if check (Line 23) and the else block (Lines 25 to 28). Such observation and simplification do not always hold as shown in the next example in § 5.1.2.

Once $S(red, 1, 3)$ is removed, $\bowtie_2$ calls its getNext() (Line 29). Since *MatchingTuples* $= [(red, 3, 2)]$ now, $(red, 3, 2)$ is returned (Line 10). r_{outer} in $\bowtie_1$ is updated to $(red, 3, 2)$ due to Line 20. Then, $\bowtie_1$ looks up $\mathcal{H}_B$ (Line 15) and computes the first and the only join result $(red, 3, 2)$ (Line 18).

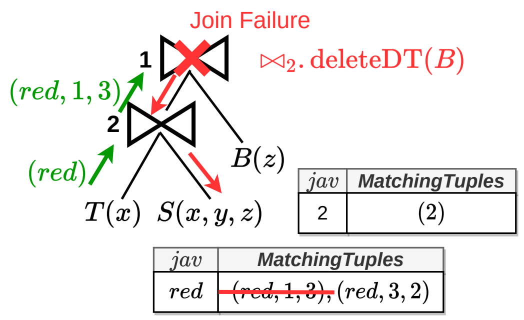

Figure 3: Join failure on $\bowtie_1$ triggers a deleteDT() **call that removes $(red, 1, 3)$ from the hash table built on S.**

5.1.2 Arbitrary Join Tree Query. Arbitrary join tree means a node in $\mathcal{T}_Q$ can have more than one child. For this third example, the 3-way join of § 5.1.1 is expanded to 4-ways: $T(x)$, $S(x, y, z)$, $B(z)$, and $R(y, z)$. The database is updated as follows: $T(green)$, $T(green)$, $T(red)$, $T(red)$, $S(red, 1, 2)$, $S(red, 3, 2)$, $B(2)$, and $R(3, 2)$. Figure 2 gives the $\mathcal{T}_Q$ of the query. In particular, S has two children: B and R. Runtime behaviors of TTJ are illustrated in Figure 4 and Figure 5.

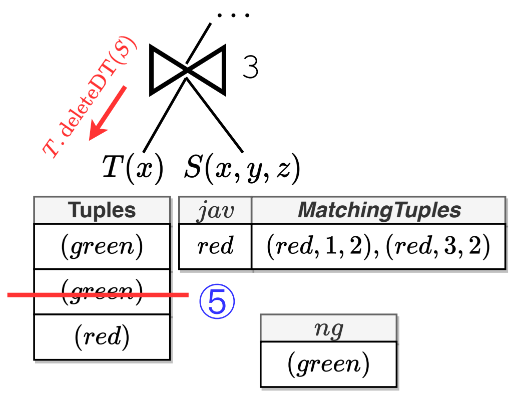

Figure 4: Critical execution moment in TTJ **plan.** $T(green)$ **is a dangling tuple and its** *jav* $x = green$ **is added to** *ng* **after receiving** deleteDT() **call from** S**.**

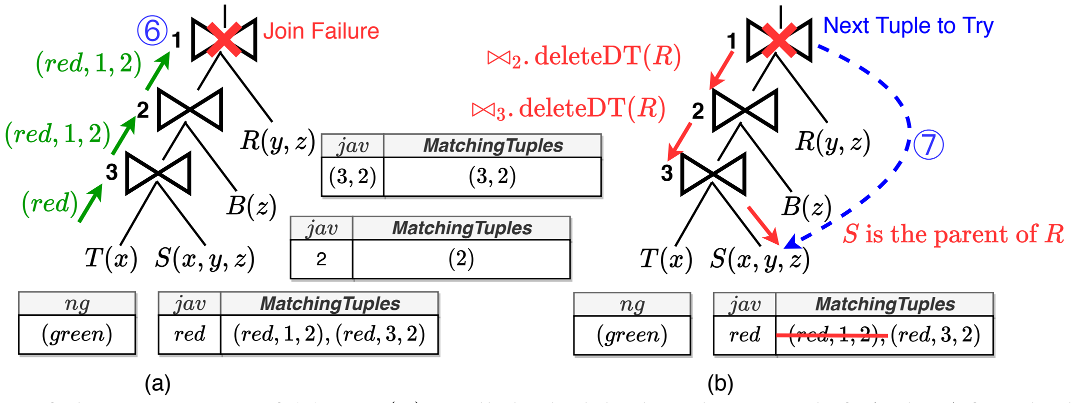

Figure 5: Join fails at $\bowtie_1$. A series of $\text{deleteDT}(\text{R})$ is called, which leads to the removal of $S(red, 1, 2)$ from hash table $\mathcal{H}_S$.

⑤ in Figure 4. After plan evaluation begins, $\text{getNext}()$ makes recursive calls to itself ending with a call to T's TTJ table scan operator (Line 4 Algorithm 5.2), which returns $T(green)$. The *jav* $x = green$ is used to look up $\mathcal{H}_3$ (Line 15). Since none of the tuples in S can join with $T(green)$, Line 20 is called. For $\bowtie_3$, R_{outer} references T and R_{inner} references S. Thus, argument to the call is S and $T.\text{deleteDT}(\text{S})$ is called, i.e., Algorithm 5.2 $\text{deleteDT}()$ method is invoked, which results in *jav* $x = green$ being added to *ng* (Algorithm 5.2 Line 10). $\text{getNext}()$ is called (Algorithm 5.2 Line 11). The second $T(green)$ is skipped because its *jav* is in *ng* (⑤). The next tuple $T(red)$ passes the *ng* check and is returned. Execution now backs to Line 20 with $r_{outer} = T(red)$.

Algorithmic Element 4. *Algorithm 5.2 is only used for R_k. Thus, $\bowtie_k$ (table scan for R_k) is the only place where ng is explicitly maintained and checked. The reason is that R_k is a base table instead of a hash table; the operator cannot directly remove tuples from R_k. Thus, the use of ng is to mimic deleting dangling tuples from a hash table that would otherwise have been built from R_k. Analogously, checking ng (Line 6 Algorithm 5.2) is to ensure that the deleted tuple never appears again.*

⑥ in Figure 5. $\bowtie_3$'s r_{outer} is $T(red)$ and its *jav* $x = red$ is looked up inside $\mathcal{H}_S$ (Line 15). *MatchingTuples* is $[(red, 1, 2), (red, 3, 2)]$. Thus, *aMatchingTuple* equals $(red, 1, 2)$ (Line 17). Then, $(red, 1, 2)$ is returned from $\bowtie_3$ (Line 18). The value of r_{outer} in $\bowtie_2$ is $(red, 1, 2)$ by Line 13. Since $B(2)$ and $(red, 1, 2)$ are joinable, *MatchingTuples* is $[(2)]$ from Line 15. $(red, 1, 2)$ is then returned from $\bowtie_2$ (Line 18). r_{outer} value of $\bowtie_1$ is now $(red, 1, 2)$ (Line 13). Looking up tuples in $\mathcal{H}_R$ of $\bowtie_1$ returns *nil*; join fails at $\bowtie_1$ (⑥).

⑦ in Figure 5. $(red, 1, 2)$ causes the join failure. Since R_{outer} is $\bowtie_2$ and R_{inner} is R, $\bowtie_1$ calls $\bowtie_2.\text{deleteDT}(\text{R})$ (Line 20). Inside $\bowtie_2$'s $\text{deleteDT}()$ implementation, since B is not the parent of R in $\mathcal{T}_Q$ (Line 23), $\text{deleteDT}(\text{R})$ is recursively called from Line 27 as $\bowtie_3.\text{deleteDT}(\text{R})$. $\bowtie_3$'s R_{inner}, S, is the parent of R in $\mathcal{T}_Q$. The algorithm removes the last returned tuple from *MatchingTuples*, which is $S(red, 1, 2)$ (Line 24, ⑦).

$\text{getNext}()$ is called from Line 29 inside $\bowtie_3$'s $\text{deleteDT}()$ method. Since *MatchingTuples* still contains $(red, 3, 2)$, $(red, 3, 2)$ is returned as the join result from $\bowtie_3$ (Line 10). In $\bowtie_2$, r_{outer} is set to $(red, 3, 2)$ by Line 27. Then, $\text{getNext}()$ is called from Line 29. $\mathcal{H}_B$ is looked up again because *MatchingTuples* is *nil* (Line 26). *MatchingTuples* is $[(2)]$ (Line 15). Join result $(red, 3, 2)$ is returned from Line 18. $\bowtie_1$'s r_{outer} is set to $(red, 3, 2)$ by Line 20. After $\mathcal{H}_R$ lookup, *MatchingTuples* is $[(3, 2)]$ (Line 15), which leads to the join result of the query $(red, 3, 2)$.

Algorithmic Element 5. *For a given join failure, MatchingTuples is set to nil on all the operators (e.g., $\bowtie_2$) that are on the path between the detection operator (e.g., $\bowtie_1$) and the removal operator (e.g., $\bowtie_3$) because r_{outer} will be changed and all the existing tuples in MatchingTuples will be invalid.*

When $\bowtie_1$'s $\text{getNext}()$ is called again, $T(red)$ is returned to $\bowtie_2$. Since $(red, 1, 2)$ is removed from S, $(red, 3, 2)$ is returned from $\bowtie_3$, which subsequently leads to the second join result of the query.

5.2 Correctness of TTJ

For the correctness and worst-case runtime analysis, we assume *ng* in Algorithm 5.2 cannot filter out any tuples from R_k; *ng* can only guarantee that a dangling tuple will never reappear once it is added to *ng*. Thus, Algorithm 5.2 Line 6 can be viewed as an implementation optimization that does not matter in the formal analysis. Further, let *iter* be an iterator on *MatchingTuples*, i.e., when calling $\text{next}()$ on *MatchingTuples*, *iter* is advanced and returns the next tuple in *MatchingTuples* if such tuple exists and *nil* otherwise.

Lemma 2. *For every value assignment to r_{outer}, MatchingTuples is initialized with tuples from $\mathcal{H}$ and implicitly, iter is reset. Between each pair of value assignments to r_{outer}, MatchingTuples is never initialized and iter is never reset.*

Proof. r_{outer} is assigned in four places: Lines 11, 13, 20, and 27. For Lines 11, 13, and 20, *MatchingTuples* is initialized on Line 15. For Line 27, since *MatchingTuples* is set to *nil* (Line 26), *MatchingTuples* is initialized on Line 15 as well. Since *MatchingTuples* is never initialized with tuples from $\mathcal{H}$ in the rest of Algorithm 5.1, the claim follows. □

Lemma 3. *A tuple, t, is part of the final join result if and only if it is not marked as dangling during the query evaluation by* $\text{deleteDT}()$*, i.e., removed from a hash table or added to ng.*

Proof. We prove the equivalent statement: a tuple t is marked as dangling by $\text{deleteDT}()$ during the query evaluation if and only if t is not part of the final join result. Whenever deleteDT() is called, a tuple is removed from a hash table or added to ng. delete DT() is initiated if and only if $MatchingTuples = nil$, which means r_{outer} contains a dangling tuple, i.e., some tuple is not part of the final join result. □

*Theorem 4 (**Correctness of** TTJ).* Executing P_Q, a left-deep query plan satisfying the default join order assumption for a full acyclic conjunctive query of k relations using $k-1$ instances of Algorithm 5.1 as join operators and 1 instance of Algorithm 5.2 as the table scan operator for R_k, computes the correct query result.

Proof. We show $J_1 = J_1^*$ under bag semantics. We first show $J_1 \subseteq J_1^*$. Let $t \notin J_1^*$. Recall

$$J_1^* = \left\{t \text{ over } sort(\mathcal{P}_{\bowtie_1}) \mid t[sort(R_u)] \in R_u \ \forall u \in [k]\right\}.$$

If there doesn't exist a relation R in Q such that $t[sort(R)] \in R$, it is trivial to see that $t \notin J_1$. Suppose $t[sort(R_u)] \in R_u$ for $u = \{k, k-1, \ldots, i+1\}$ but $t[sort(R_u)] \notin R_u$ for $u = \{i, i-1, \ldots, 1\}$. By default join order (Definition 1), R_i must be a child of some relation R_j with $i < j$ such that $t[sort(R_j)] \in R_j$ and $t[sort(R_i)] \notin R_i$. By $\mathcal{T}_Q$ definition, $sort(R_i) \cap sort(R_j) \neq \emptyset$. The only non-trivial reason that $t[sort(R_i)] \notin R_i$ is because $t[sort(R_i) \cap sort(R_j)] \notin \pi_{sort(R_i) \cap sort(R_j)}(R_i)$. In such case, TTJ will call deleteDT() from the join operator connected with R_i and $t[sort(R_j)]$ will be deleted from $\mathcal{H}_{R_j}$ or put onto ng. Thus, t is not in J_1. If there is a relation R_u with $k \leq u \leq i+1$ such that $t[sort(R_u)] \notin R_u$, $t \notin J_1$ by the definition of join. The same argument applies if t are duplicated.

To show $J_1^* \subseteq J_1$, suppose $t \in J_1^*$ but $\notin J_1$. $t[sort(R_1)]$ is part of the join result and with Lemma 3, $t[sort(R_1)]$ is never deleted. Thus, it must be that $t[sort(\mathcal{P}_{\bowtie_2})] \in J_2^*$ but $t \notin J_2$. The same argument applies to every operator in the plan. Eventually, we have $t[sort(R_k)] \in J_k^*$ but $t \notin J_k$. However, this is a contradiction. $t[sort(R_k)] \in J_k^*$ and joins with the rest of the relations in plan. Thus, with Lemma 3, $t[sort(R_k)] \notin ng$ and $\in J_k$.

Next, we show $|J_1| = |J_1^*|$. That is, for a given $t \in J_1^*$, we show the number of tuples t that are in J_1^* equals to the number of tuples t in J_1. With Lemma 3, TTJ will not falsely remove a tuple t that is in J_1^* and if t is a dangling tuple, it is removed by deleteDT( ). Further, by Lemma 2, each tuple from $\bowtie_u \bowtie R_{u-1}$ is enumerated once. Thus, the claim holds. □

6 OPTIMALITY OF TTJ

The complexity analysis of a collection of $k-1$ TTJ operators to compute a k-way join, Q, is done in two parts. We first propose a general condition for any left-deep plan $\mathcal{P}_Q$ satisfying the default join order assumption called the *clean state* (§ 6.1). Clean state is an intermediate point in the evaluation of the $\mathcal{P}_Q$ where we prove that from the clean state the time required to complete the query evaluation is $\mathcal{O}(n+r)$. We then show P_Q is able to reach the clean state and the work done by TTJ between beginning of the query evaluation and reaching the clean state is no more than the work done after reaching the clean state (§ 6.2).

6.1 Clean State

Any full CQs (not just acyclic CQs) can be evaluated in $\mathcal{O}(n+r)$ if the relations have no dangling tuples. One way to achieve such goal for acyclic CQs is *full reducer* F_Q [11]. A forklore result [2] states that using *reducing semijoin program* HF_Q [11] is sufficient for the aforementioned complexity. Furthermore, HF_Q allows the existence of dangling tuples. However, even HF_Q removes more dangling tuples than necessary. We define a more relaxed condition, *clean state*, to represent the state of relations with the presence of dangling tuples but still admits the optimal evaluation. We discuss the relationship among F_Q, HF_Q, and clean state in more details after Theorem 6.

*Definition 2 (**clean state**).* $\mathcal{P}_Q$ satisfying Definition 1 reaches *clean state* if contents of the relations in the plan satisfies the following conditions:

(i) $\mathbb{R}_k \overline{\ltimes} \widetilde{\mathbb{R}}_u = \emptyset$ for the root of $\mathcal{T}_Q$, R_k and its children R_u;

(ii) $\mathbb{R}_i = \widetilde{\mathbb{R}}_i$ for all the leaf relations R_i of $\mathcal{T}_Q$; and

(iii) $(\mathbb{R}_i \ltimes J_{i+1}^*) \overline{\ltimes} \widetilde{\mathbb{R}}_u = \emptyset$ for internal° relation R_i and their child relations R_u. The content of R_i satisfying the condition is denoted by $\widetilde{\mathbb{R}}_i$

Our key result in this section is to show clean state enables optimal evaluation, i.e., once $\mathcal{P}_Q$ reaches clean state, $\mathcal{P}_Q$ evaluation will not produce dangling tuples and optimal.

Lemma 5. *When $\mathcal{P}_Q$ is in clean state, R_k is the same as R_k^*.*

Proof. Suppose $\mathcal{P}_Q$ is in clean state. Assume there is a dangling tuple $d \in R_k$. Suppose $\{d\} \bowtie R_{k-1} \bowtie \ldots \bowtie R_j$ but cannot join with R_{j+1} with $j \in \{k-1, \ldots, 2\}$. Given $\mathcal{P}_Q$ satisfying Definition 1, parent of R_{j+1}, R_i, must be one of the relations joinable with $\{d\}$. Thus, R_i is not in clean state. Contradiction. □

*Theorem 6 (**Clean state implies optimal plan evaluation**).* If $\mathcal{P}_Q$ is in clean state, any intermediate result generated from $\mathcal{P}_Q$ evaluation will not contain dangling tuples and $\mathcal{P}_Q$ can be evaluated optimally.

Proof. Proof by induction on the height of $\mathcal{T}_Q$, h. Base case $h = 0$. Claim trivially holds. Suppose the claim holds for height of $\mathcal{T}_Q < h$. Let R_h be the root of $\mathcal{T}_Q$ with height h. Let R_j be a child of R_h. With Lemma 5, no dangling tuples produced when R_h join with R_j. By induction assumption, no dangling tuple produced when further join $R_h \bowtie R_j$ with relations in subtree rooted in R_j. Repeat the same argument for each child of R_h and the result follows. Notice the order of R_js that invoke proof arguments is specified by the order in $\mathcal{P}_Q$, which satisfies Definition 1. □

Discussion. HF_Q is a sequence of semijoins and its order is determined by traversing $\mathcal{T}_Q$ bottom-up and constructing $R_p \ltimes R_c$ where R_p is a parent relation and R_c is one of its children. The resulting relations after HF_Q are denoted R_i'. F_Q extends HF_Q by traversing $\mathcal{T}_Q$ top-down applying $R_c' \ltimes R_p'$ ($R_c \ltimes R_p'$ if R_c is a leaf node). After F_Q, relations are free of dangling tuples, i.e., $\mathbb{R} = \mathbb{R}_i^*$ for $i \in [k]$. Clearly, relations that are free from dangling tuples are in clean state. Thus, relations after full reducer F_Q are in clean state. Relations after HF_Q are in clean state as well. Leaf relations after HF_Q satisfy Condition (ii) (by definition of HF_Q) and the root relation after HF_Q satisfies Condition (i) (by Lemma 5 and Lemma 4 of [11]). For an internal° relation R_i, it satisfies $\mathbb{R}_i \overline{\ltimes} \widetilde{\mathbb{R}}_u = \emptyset$, which implies the satisfaction of Condition (iii). Evidently, state of relations after HF_Q or F_Q is stricter than what is required by clean state, i.e., removing more tuples than necessary for the optimal evaluation; tuples of R_i that are not joinable with J^*_{i+1} will be removed by both F_Q and HF_Q if such tuples are not joinable with tuples from any child relation of R_i. However, those tuples are allowed to present in clean state. F_Q has its own advantages over clean state and HF_Q by not relying on join order, i.e., optimality of both clean state and HF_Q requires relations to be joined as the top-down pass of $\mathcal{T}_Q$, which is not needed by F_Q due to the absence of dangling tuples.

6.2 Runtime Analysis of TTJ

Lemma 7. *Algorithm 5.2 Line 10 is executed whenever* $\mathbb{R}_k \overline{\ltimes} \mathbb{R}_u \neq \emptyset$ *for child relation* R_u *of* R_k*. Similarly, Line 24 is executed whenever* $\mathbb{R}_i \overline{\ltimes} \mathbb{R}_u \neq \emptyset$ *for internal*$^\circ$ *relations* R_i *and its child* R_u*.* $\mathbb{R}_u$ *indicates the content of* R_u *can change during* TTJ *execution.*

Proof. We prove the claim on Algorithm 5.2 Line 10; claim on Line 24 can be proved similarly. $t \in R_k$ can be dangling for two reasons: (1) t is dangling at the very beginning of the execution, i.e., $\{t\} \overline{\ltimes} R_u = \{t\}$. Then, during the execution with t from $\bowtie_k$, join fails at $\bowtie_u$, and deleteDT() is initiated (Line 20). Since R_k is the parent of R_u, Algorithm 5.2 Line 10 is executed. (2) t becomes dangling after all tuples from $R_u \ltimes \{t\}$ are removed. After the last tuple in $R_u \ltimes \{t\}$ is removed by Line 24, *MatchingTuples* becomes empty at $\bowtie_u$. Line 29 is then called. Since *MatchingTuples* $= \emptyset$ and $R_u \ltimes \{t\} = \emptyset$, Line 15 is executed and returns *nil*. deleteDT() is initiated and Algorithm 5.2 Line 10 will be executed. □

Lemma 8. P_Q *is in clean state once* TTJ *finishes execution.*

Proof. *Satisfaction of Condition* (ii). Suppose R_i is a leaf relation. Since relations that have tuples removed or put into *ng* are parent of some other relations in $\mathcal{T}_Q$, condition holds.

Satisfaction of Condition (iii). Start with internal$^\circ$ relations R_i that are parent of leaf relations R_u. Then, $\mathbb{R}_u = \widetilde{\mathbb{R}}_u$. By Lemma 3 and parent-child relation between R_i and R_u, $(\mathbb{R}_i \ltimes J^*_{i+1}) \overline{\ltimes} \widetilde{\mathbb{R}}_u$ is empty. Thus, $\mathbb{R}_i = \widetilde{\mathbb{R}}_i$ when TTJ finishes execution. Now, let R_i be an internal$^\circ$ relation and R_u be its child, which is also an internal$^\circ$ relation. Start R_u be the parent of leaf relations and apply the same argument from the previous case. $\mathbb{R}_i = \widetilde{\mathbb{R}}_i$. Repeat the same argument all the way till R_u be the grandchild of R_k.

Satisfaction of Condition (i). By Lemma 5, equivalently, we show $\mathbb{R}_k - ng = \mathbb{R}^*_k$. (1) $\mathbb{R}^*_k \subseteq \mathbb{R}_k - ng$. Suppose $t \notin \mathbb{R}_k - ng$. This means t is one of the tuples removed by Algorithm 5.2 Line 10. With Lemma 7, $t \notin \mathbb{R}^*_k$. (2) $\mathbb{R}^*_k \supseteq \mathbb{R}_k - ng$. Suppose $t \notin \mathbb{R}^*_k$. Then, t has to be a dangling tuple causes a join failure at some relation R. By the proof of Lemma 7, either deleteDT() is called directly (R is a child of R_k) or indirectly (R causes all tuples from R_u, a child of R_k, joining with t removed). Thus, $t \notin \mathbb{R}_k - ng$. □

Lemma 9. TTJ *evaluates* P_Q *in* $\mathcal{O}(n + r)$ *once* P_Q *is clean.*

Proof. By Theorem 6, once P_Q reaches clean state, no dangling tuple is produced by $\bowtie_u$ for $u \in [k]$. Thus, no more calls on delete DT(). There are k relations and $k - 1$ join operators, open() takes $O(kn)$ as each operator is called once and takes $O(n)$ to build $\mathcal{H}$. It takes $O(k)$ getNext() calls to compute a tuple in J^*_1. Since each getNext() call takes $O(1)$, it takes $O(k)$ to compute one join result and $O(kr)$ for J^*_1. Thus, in total, we have $O(kn+kr) = \mathcal{O}(n+r)$. □

Evaluating P_Q can now be viewed as containing two pieces of work: (1) *non-result-generation step* that makes P_Q reach clean state; and (2) *result-generation step* to generate the join result, which takes $\mathcal{O}(n + r)$. Thus, to show TTJ optimality, we show the cost of non-result-generation step is no larger than the result-generation step. We denote the cost associated with the non-result-generation step as *cleaning cost.*

Theorem 10 (***Data complexity optimality of*** TTJ). *k TTJ instances, $k - 1$ instances of Algorithm 5.1 for join operators and 1 instance of Algorithm 5.2 for R_k, evaluates acyclic CQs of k relations using left-deep query plan with default join order in $\mathcal{O}(n+r)$, meeting the optimality bound for acyclic CQs in data complexity.*

Proof. By Lemma 8, the execution of a plan is in clean state when TTJ execution finishes. The amount of work that makes P_Q clean, i.e., cleaning cost, is fixed despite the distribution of dangling tuples in the relations. Suppose the execution is in clean state after computing the first join result.

To bound the cleaning cost, we bound the cost of getting the first join result. Cleaning cost of TTJ includes the following components: (1) the cost of open(), which is $O(kn)$; (2) the cost of getNext(); and (3) the cost of deleteDT(), which is bounded by the cost of getNext() as well.

The total cost of getNext() is bounded by the total number of loops (starting at Line 14). Within the loop, hash table lookup (Line 15) is $O(1)$. The total number of loops equals to the total number of times that r_{outer} is assigned with a value. r_{outer} assignment happens on Lines 11, 13, 20, and 27. Line 13 is called when get Next() is recursively called from $\bowtie_1$ to start computing the first join result, which in total happens k times. Afterwards, whenever r_{outer} becomes *nil*, execution terminates by returning *nil* (Lines 12, 21, and 28) and Line 13 never gets called.

Each time deleteDT() is called from Line 20, exactly one tuple is removed. Thus, r_{outer} is assigned $O(kn)$ times on Line 20. After a call to deleteDT() made in the ith operator ($i \in [k-2]$) from Line 20, deleteDT() can be recursively called at most $k - i$ times from Line 27. The number of deleteDT() calls with $k - i$ recursive calls is at most n because each relation has size n and each initiation of deleteDT() removes a tuple. Thus, the total number of assignment to r_{outer} from Line 27 is $\leq \sum_{i=1}^{k-2}(k - i) \cdot n = O(k^2 n)$.

If deleteDT() is never called during the computation of the first join result, Line 11 is not called. Line 11 can only be called from Line 29 when Line 23 is evaluated to true; any getNext() calls (Line 29) from recursive deleteDT() calls triggered by Line 20 will not call Line 11 because *MatchingTuples* is set to *nil* on Line 26. Thus, the number of calls on Line 11 equals to the number of dele teDT() calls from Line 20, which is $O(kn)$.

Summing everything together, cleaning cost is $O(k^2 n)$. Since P_Q is clean after computing the first join result, with Lemma 9, the result follows. □

Theorem 10 gives $O(k^2 n + kr)$ combined complexity. We can further improve it to $O(nk \log k + kr)$ by constraining the default order assumption (Definition 1). In particular, to decide join order, one pre-order traverses $\mathcal{T}_Q$ and when multiple subtrees exist for a given relation in $\mathcal{T}_Q$, one breaks ties by visting the largest subtree of any relation last [8]. Figure 6 shows an example.

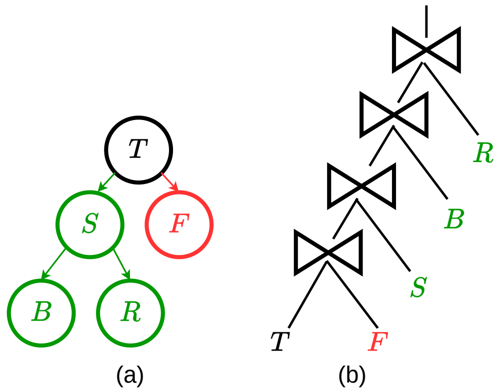

Figure 6: Given $\mathcal{T}_Q$ in (a), one decides join order by pre-order traversing over $\mathcal{T}_Q$ and breaking ties via visiting the largest subtree of any relation last. The resulting order (b) satisfies default join order assumption.

*Theorem 11 (**Improving combined complexity of** TTJ).* Combined complexity of TTJ can be improved to $O(nk \log k + kr)$ (log is base 2) if one performs pre-order traversal over $\mathcal{T}_Q$ and break ties by visiting the largest subtree of any relation last.

Proof. The new order strategy only changes the total number of $\mathrm{deleteDT}()$ calls (Line 27) in Theorem 10 proof. For a given $\mathcal{T}_Q$, let b_i denote the number of relations between the join failure relation R_i and its parent. Note that b_i is exactly the same as the number of $\mathrm{deleteDT}()$ calls generated (Line 27) when join fails at $\bowtie_i$. $b_i \le k - i$ for the default join order. Let d_i denote the number of descendents of an internal relation R_i and m_i denotes the number of relations in the largest subtree rooted at one of i's children, e.g., in Figure 6, $d_T = 4$ and $m_T = 3$. Since the new order satisfies Definition 1, when join fails at R_i, only descendents of R_j (the parent of R_i) could exist between R_i and R_j in the order. The largest number of $\mathrm{deleteDT}()$ generated when join fails at the root relation of the largest subtree of a relation. Thus, $b_i \le d_i - m_i + 1$.

Next, we prove $\sum_{i=1}^{k-1} b_i \le k \log k$. Proof by induction on the size of $\mathcal{T}_Q$. Base case $k = 1$, the claim holds. Assuming the claim holds for $k - 1$. Suppose there are s subtrees of R_k and each with size $k_1, \ldots, k_s$. Let k_m denote the largest subtree. Then $b_r \le (k - 1) - k_m + 1 = k - k_m$. Thus, $\sum_{i=1}^{k-1} b_i \le \sum_{i=1}^{s} k_i \log k_i + (k - k_m) \le k \log k_m + (k - k_m) \le k \log k$ (the last inequality follows Lemma A.1 in [8]). Then, the total number of $\mathrm{deleteDT}()$ calls on Line 27 is $\le \sum_{i=1}^{k-1} b_i n = O(nk \log k)$. □

7 TWO PRACTICAL CONSIDERATIONS

To put TTJ into practice, there are issues beyond the algorithm itself. Two of the most important ones are: (1) how to model the cost of TTJ and enable cost-based optimization (§ 7.1); and (2) how to use TTJ with bushy query plans (§ 7.2).

7.1 TTJ Cost Model

To determine a good P_Q the cost must consider and model the use of $\mathcal{T}_Q$. One simple but effective cost model is the sum of intermediate result sizes [13, 15, 30, 46], which comprises two components: the dangling tuples produced and the size of intermediate results that are part of final join result. Thus, the idea of the model is to estimate the impact of TTJ on those components. Since TTJ also reduces inner relation sizes, like [15], TTJ cost includes the size of inner relations that are in clean state as well. Theorem 12 gives the TTJ cost equation.

Theorem 12. The cost of TTJ, i.e., the cost of $\mathcal{T}_Q$ when evaluated by TTJ, is

$$\sum_{i=1}^{m} \sum_{t=0}^{|\mathcal{A}^i|-1} b^{R_i}_{R_u^{t+1}} |(\mathbb{R}_i^{[t]} \ltimes J^*_{i+1}) \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u^{t+1})| \tag{1}$$

$$+ \sum_{i=1}^{s} \sum_{t=0}^{|\mathcal{B}^i|-1} b^{R_i}_{R_u^{t+1}} |(\mathbb{R}_i^{[t]} \ltimes J^*_{i+1}) \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u^{t+1})| \tag{2}$$

$$+ \sum_{t=0}^{|C|-1} b^{R_k}_{R_u^{t+1}} |\delta(\pi_{ja(R_k, R_u^t)} (\mathbb{R}_k^{[t]} \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u^{t+1}))| \tag{3}$$

$$+ \sum_{j=k}^{1} |f(S_j)| \tag{4}$$

$$+ \sum_{i=2}^{k} |\widetilde{\mathbb{R}_i}| \tag{5}$$

Equations (1) to (3) give the number of dangling tuples, which are built around clean state. Equation (4) gives the size of intermediate results (including the size of R_k^*) that are part of final join result. Equation (5) is the summation of size of internal° and leaf relations that are in clean state.

As defined in the proof of Theorem 11, b_S^R is the number of relations between R and S in P_Q, which counts the number of additional dangling tuples are generated given a dangling tuple from R and detected at S. We define the following three sets over the relations in $\mathcal{T}_Q$: (1) $\mathcal{A}$ consists of all the leaf relations R_u such that internal° relation R_i are their parent. We partition $\mathcal{A}$ by leaf relations' parents, $\mathcal{A}^1, \ldots, \mathcal{A}^m$ where $\mathcal{A}^i$ is the set of leaf relations that have the parent R_i. Thus, $|\mathcal{A}^i|$ represents the number of leaf relations in $\mathcal{T}_Q$ that are children of R_i. Let us label those leaf relations $R_u^1, \ldots R_u^{|\mathcal{A}^i|}$; (2) $\mathcal{B}$ consists of internal° relations R_u that their parents R_i are internal° relations. Similarly to $\mathcal{A}^1, \ldots, \mathcal{A}^m$, we partition $\mathcal{B}$ by the parent of R_u: we have $\mathcal{B}^1, \ldots, \mathcal{B}^s$. $|\mathcal{B}^i|$ and $R_u^1, \ldots, R_u^{|\mathcal{B}^i|}$ are defined similarly as above; and (3) C comprises all the relations R_u that are children of R_k. The children of R_k are labeled $R_u^1, \ldots, R_u^{|C|}$. Equation (6) defines $\mathbb{R}_i^{[t]}$, which reflects the gradual discovery of dangling tuples of R_i during P_Q evaluation.

$$\mathbb{R}_i^{[t]} = \begin{cases} \mathbb{R}_i & \text{if } t = 0 \\ \mathbb{R}_i^{[t-1]} - ((\mathbb{R}_i^{[t-1]} \ltimes J^*_{i+1}) \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u^t) & \text{otherwise} \end{cases} \tag{6}$$

Suppose the join order (w.r.t. $\mathcal{T}_Q$) determined either syntactically from $\mathcal{T}_Q$ or from the DP algorithm is $[S_k, \ldots, S_1]$ where $S_j = R_i$ for some $i \in [k]$ (the order is from the left to right in P_Q). $f(S_j)$

in Equation (7) computes the size of intermediate results that are part of the final join result. Given Definition 1, the first relation to apply f is always R_k, root of $\mathcal{T}_Q$, and its content, per Lemma 5, is $\mathbb{R}_k^*$.

$$f(S_j) = \begin{cases} \mathbb{R}_k^* & \text{if } j = k \\ \widetilde{\mathbb{S}}_j \bowtie f(S_{j+1}) & \text{otherwise} \end{cases} \tag{7}$$

Example 2. Using the example from § 5.1.2, we illustrate how to compute Equations (1) to (3), the most complex terms in the cost equation. Assuming the join order is $[T, S, B, R]$. Start with Equation (1). $\mathcal{A} = \{B, R\}$. Since both B and R have the same parent, S, $m = 1$. Thus, the cost is $1 \cdot |(S \ltimes T) \overline{\ltimes} B| + 2 \cdot |(S^{[1]} \ltimes T) \overline{\ltimes} R|$. In particular, $1 \cdot |(S \ltimes T) \overline{\ltimes} B| = 0$. Therefore, $S^{[1]} = S$. $2 \cdot |(S^{[1]} \ltimes T) \overline{\ltimes} R| = 2$ due to $S(red, 1, 2)$. Thus, $\widetilde{S} = \{(red, 3, 2)\}$. Since $\mathcal{B} = \emptyset$, we do not need to compute Equation (2). To compute Equation (3), due to $C = \{S\}$, we have $1 \cdot |\delta(\pi_x(T \overline{\ltimes} \widetilde{S}))| = 1$. Thus, the number of dangling tuples produced by TTJ is $0 + 2 + 1 = 3$.

Our empirical evaluation implements the cost model in combination with the DP algorithm to decide both $\mathcal{P}_Q$ and $\mathcal{T}_Q$ for TTJ (detailed in § 8.1). A limitation of Theorem 12 is the lack of physical cost coefficients. Addressing such limitation is a future work.

7.2 Bushy Plan

A common approach to evaluate a bushy plan is to decompose it into a sequence of left-deep subplans: right child of every join operator forms a left-deep subplan and is evaluated first before proceeding with the join [45, 53]. In particular, for in-memory hash join, build side is a blocking operation, i.e., hash tables can be constructed not just from base relations but also from intermediate results computed from subplans, which are buffered inside the memory [25, 45]. TTJ works with bushy plan exactly as above. The only issue to address is to have the bushy plan satisfy the default join order assumption (Definition 1) so that when join fails at R, deleteDT() can find its parent. We use Algorithm 7.1 to control the construction of a bushy plan for TTJ. Such algorithm can be easily adapted into a decision procedure to check compatibility of a given bushy plan with TTJ, i.e., constructing a $\mathcal{T}_Q$ from the given plan such that default join order assumption is satisfied.

Fragment group FG is a set of nodes in $\mathcal{T}_Q$ constituting a subplan. We use FG and subplan interchangeably. Any node from $\mathcal{T}_Q$ only belongs to one group. The key idea to form a bushy plan is that we create a TTJ-compatible subplan for each group and connect them altogether using TTJ join operators again. Fragment groups are formed with the property that parent node belongs to the same or lower-numbered group than its child node(s) in $\mathcal{T}_Q$. Line 2 checks sibling node to avoid cross-product when join two subplans. The resulting plan satisfies the default join order assumption and can be evaluated by TTJ directly.

Example 3. Consider Q represented in Figure 7 (a). There are two fragment groups $FG_1 = \{T, S, R\}$ and $FG_2 = \{U, D, V\}$. The whole plan is being evaluated by TTJ operators: every join operator is TTJ join operator; T and U are TTJ table scan operators and the rest are normal table scan operators. deleteDT() happens for two cases: (1) deleteDT() happens inside the subplan. For example, join fails at $\bowtie_4$ (⑧). Since U is the parent of V, a tuple from U is added to ng; and (2) deleteDT() happens at the join operator connecting two subplans. For example, join fails at $\bowtie_1$ (⑨). In this case, deleteDT() sends the reference to the root of FG_2, U, downward. The rest of the evaluation is the same as the left-deep plan case in the previous sections.

Algorithm 7.1: Construct bushy plan for TTJ

```
Input: $\mathcal{T}_Q$
Output: A bushy plan that can be evaluated by TTJ
1 Starting from the root of $\mathcal{T}_Q$, visit each node in pre-order
    traversal.
2 For each node, decide whether create a new fragment
    group $FG_{i+1}$ or put it to the fragment group $FG_i$ where its
    parent node belongs. If for a node $R$ and its left sibling
    node $S$ has $sort(R) \cap sort(S) = \emptyset$, $R$ has to be in the same
    group as its parent. Suppose there are fragment groups
    $FG_i$ for $i \in [m]$ at the end of this step.
3 For each $FG_i$ for $i \in [m]$, create a subplan satisfying the
    default join order .
4 Start from $FG_m$ and connect it with the subplan from
    $FG_{m-1}$ with a TTJ join operator. The resulting subplan, a
    new $FG_{m-1}$, is connected with the subplan $FG_{m-2}$ and
    continue. When connecting two subplans $FG_i$ and $FG_{i-1}$,
    we always put $FG_{i-1}$ as the left child of TTJ join operator.
    The step repeats until all the subplans are connected.
```

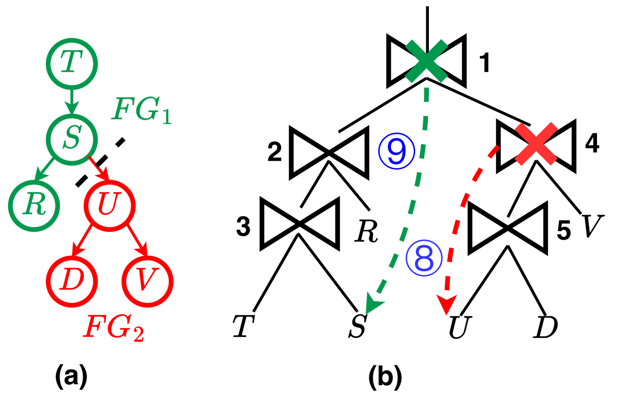

Figure 7: Given $\mathcal{T}_Q$ in (a), there are two fragment groups FG_1 and FG_2. (b) is a bushy plan constructed from the two fragment groups. Join failures can be categorized into two cases: within FG (⑧) and across FGs (⑨).

Lemma 13. *The bushy plan constructed from Algorithm 7.1 satisfies default join order assumption (Definition 1).*

Proof. Let S be a node and R, U be its children. There are three possible cases: (1) if R and U are all within the same FG as S, by Line 3, the claim holds; (2) if one of its children is in a different FG, say, U. Since S is in the FG with smaller numbering, by Line 4, S is to the left of U. S is to the left of R because they are in the same FG; and (3) if all of its children are in different FGs, by a similar argument as the previous case, the claim holds. □

*Theorem 14 (**Correctness**).* TTJ evaluates Q correctly under the bushy plan constructed from Algorithm 7.1.

Proof. By Lemma 13 and Theorem 4, TTJ evaluates relations associated with each FG correctly. We only need to focus on TTJ operators that join two different FGs and show it generates the correct join result. W.l.o.g., the two FGs are denoted FG_1 and FG_2. We want to show $FG_1 \bowtie FG_2$ is correct. If all tuples from FG_1 are joinable with some tuple from FG_2 join result, the claim holds. Suppose $t \in FG_1$ cannot join with any tuple from FG_2 and thus dangling. By Lemma 13, the parent of the node where join fails must be in FG_1. Applying the same arguments in Theorem 4, the claim holds. □

Let r_i denote the size of the join result computed from FG_i.

*Theorem 15 (**Data complexity of** TTJ **on bushy plan**).* Suppose there are m FGs and each has result size $r_1, \ldots, r_m$, respectively. TTJ runs $\mathcal{O}(n + \max(r_1, \ldots, r_m))$.

Proof. Note that $m \leq k$. Proof by induction on the number of FGs in the plan. Base case FG_m. It takes $\mathcal{O}(n + r_m)$ to evaluate it. Suppose the claim holds for all the fragment groups until i. To evaluate the plan associated with FG_{i-1}, it takes $\mathcal{O}(n + \max(r_{i+1}, \ldots, r_m))$ to evaluate the subplan associated with FG_i. By Line 4, FG_i appears as r_{inner}, which TTJ builds the hash table. TTJ takes r_i to build $\mathcal{H}$ corresponding to FG_i. Result follows. □

8 EVALUATION

Our empirical study uses HJ as a baseline against which we can evaluate how TTJ performs on acyclic CQs.

8.1 Experimental Setup

Implementation. We implemented a query engine in Java. The design is similar to the architecture of recent federated database systems [10, 43]. The query optimizer uses the standard DP algorithm [18] to determine both join orders (for HJ and TTJ) and join tree (for TTJ). The optimizer only considers left-deep query plans without cross products. Thus, by Lemma 1, each cell of the DP matrix guarantees the existence of at least one partially-built $\mathcal{T}_Q$ and the resulting join order satisfies the default join order assumption (Definition 1). The best $\mathcal{T}_Q$ associated with each cell is determined by Theorem 12. Thus, the optimization step does not enforce special join order in Theorem 11. The query engine uses the PostgreSQL 13 optimizer to provide estimation to the relational algebra terms in Theorem 12 cost equation. Specifically, we load a PostgreSQL instance with benchmark data. For each term (without b_S^R) in the TTJ cost equation, SQL statements, embedded as EXPLAIN commands, are generated and sent to the PostgreSQL instance. The estimation is then extracted from the returned result. For HJ, the cost function is the sum of intermediate result sizes. The estimations to the HJ cost equation are determined the same way as TTJ. Since the cost functions of HJ and TTJ are different, the optimizer may determine different join orders for each algorithm on the same query. The query engine uses DuckDB [38] as the storage manager because of its low data fetching overhead. Select predicates are pushed down and evaluated by DuckDB by accessing data through non-materialized views. Thus the evaluation of the queries is identical across algorithms except for the join processing.

Environment. For all our experiments, we use a single machine that has one AMD Ryzen 9 5900X 12-Core Processor @ 3.7Hz CPU and 64 GB of RAM. We only use one logical core. We set the size of the JVM heap to 20 GB. All the data structures (e.g., hash tables, *ng*) are stored in JVM heap. We use JMH [1] to orchestrate benchmarking with 5 warmup forks and 10 measurement forks. Within each fork, there are 3 warmup iterations and 5 measurement iterations. We omit optimization time since it was dominated by the JDBC overhead of using the PostgreSQL optimizer to compute basic information.

Table 1: Performance of TTJ relative to HJ on all 113 JOB queries.

Perf. Impr. %	Count	JOB Queries
[35%, 40%)	1	10c
[25%, 30%)	2	3b,19d
[20%, 25%)	5	22a,22b,22d,28b,28c
[15%, 20%)	6	9d,19c,22c,28a,30a,31c
[10%, 15%)	1	30c
[5%, 10%)	6	4c,9c,12c,13a,15b,29a
[0%, 5%)	42	1d,2a,2b,2c,3a,3c,4a,6a,6b,6c,6d,6e,7b, 8a,8b,9b,10a,10b,12a,12b,13c,14b,14c, 15d,16a,16b,16d,17f,18a,19b,20b,21b,21c, 23b,25a,25c,26b,26c,29c,31a,33a,33b
[-5%, 0%)	46	1a,1b,1c,2d,4b,5a,5c,6f,7a,7c,8c,8d,9a, 11a,11b,11c,11d,13b,14a,15a,15c,16c,17a, 17c,17d,17e,18b,19a,20a,20c,21a,23a,23c, 24a,24b,25b,26a,27a,27b,27c,29b,30b,31b, 32a,32b,33c
[-10%, -5%)	2	5b,17b
[-15%, -10%)	1	18c
[-25%, -20%)	1	13d

Workload. We evaluate algorithms on two workloads: Join Ordering Benchmark (JOB) [30] and TPC-H [48] (scale factor = 1). We focus on the acyclic CQs in the benchmarks, i.e., we omit cyclic queries, single-relation queries, and queries with correlated subqueries. If queries are not CQ, we removed the non-CQ parts. All 113 JOB queries meet the criteria as do 13 TPC-H queries.

8.2 Query Performance

Table 2: Performance of TTJ relative to HJ on all 13 TPC-H acyclic conjunctive queries.

Performance Improvement %	Count	TPC-H Queries
[50%, 55%)	1	8
[5%, 10%)	1	7
[0%, 5%)	6	3,10,11,15,16,20
[-5%, 0%)	5	9, 12, 14, 18, 19

Table 1 shows the performance improvement percentage of JOB queries when evaluated using TTJ comparing to using HJ. Each row is similar to a bin in a histogram. The bin width is 5%. Empty bins are omitted. The red line of the table separates performance improvements from the regression: Queries listed above the red line indicates that TTJ performs better than HJ. Of 113 JOB queries, TTJ improves 63 (56%) of them. In particular, the largest improvement is 39% coming from 10c. Two significant performance regression (more than -10%) queries are 13d and 18c. Both are due to less than ideal TTJ join orders. In specific, during the DP process, PostgreSQL provides up to 60× overestimation towards Equation (4), which causes cells that lead to better orders being discarded early in the process. To ensure that we can indeed remove regression if we have more accurate estimation to the cost equation, as a workaround, we temporarily removed Equation (4) from the cost equation, which yielded different join orders for these two queries. The new join orders improve the performance of both queries: 13d improves from -21% to -11% and 18c improves from -14% to -1%.

Same as Table 1, Table 2 shows the performance improvement percentage of TPC-H queries when evaluated using TTJ comparing to using HJ. Q8 has the largest performance improvement percentage, 53%. Q8 is the largest query in terms of number of join relations, $k = 8$, in the TPC-H benchmark. Q8 result along with results of large queries in JOB (e.g., 28c has performance improvement 22%) shows that TTJ performs well on complex acyclic join queries. On the other hand, a few TPC-H queries we run are simple binary join queries. In those queries, TTJ performs on par with HJ. For example, Q12 is a binary join query and TTJ performance improvement percentage is -1%. Thus, the result from both benchmarks shows that TTJ has performance merit over HJ on acyclic CQs under proper join order.

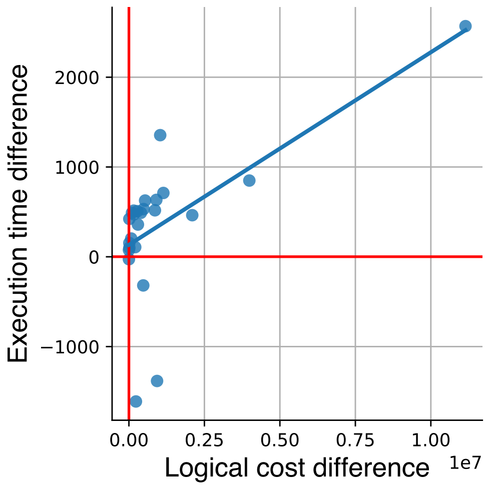

Figure 8: Execution time difference (in ms) against logical cost difference between HJ and TTJ on JOB queries with |performance improvement percentage| > 5%. The line plots the linear regression between X-axis and Y-axis.

To have an understanding of the cause of the performance difference between TTJ and HJ, we closely examined the JOB queries that performance improvement percentage is above 5% and below -5%, i.e., the queries that TTJ performs either extremely better or significantly worse than HJ. In total, there are 25 such queries. For 10c, surprisingly, we find that intermediate results produced by TTJ is 15% more than the intermediate results produced by HJ. However, the summation of base relations is reduced by 18% using TTJ. Thus, in total, TTJ has 8% less tuples in memory than HJ during query evaluation. Figure 8 illustrates the result of the analysis of all the selected queries. The logical cost consists of both summation of intermediate results and the base table sizes measured after query evaluation. From the figure we see that for the majority of the queries, 22 in total, the execution time difference grows as the cost difference increases. However, there are 3 queries where TTJ is slower than HJ despite TTJ has smaller logical cost than HJ. This likely suggests physical cost should also be considered during the costing for a real system. We leave it as the future work.

9 CONCLUSION AND FUTURE WORK

We have presented a modified HJ, TTJ, that computes both semijoin and join during the query evaluation by removing dangling tuples. The use of the iterator interface in TTJ design facilitates the possibility of easy integration with existing query systems. Formal analysis shows that TTJ is data complexity optimal for acyclic CQs. Empirically, TTJ largely outperforms HJ after proper optimization for both algorithms. In particular, TTJ has advantages over HJ in large queries. We discuss some limitations and future work.

Practical concerns. We are integrating TTJ into an existing RDBMS. That effort requires additional basic considerations beyond implementing the operator. (1) Cost model described in § 7.1 is a logical model, which is sufficient to determine proper join orders [13, 15, 30, 46]. Integration into a larger system requires a cost model that is consistent with the existing cost models in the RDBMS, which usually includes physical cost coefficients, such as the time required for hash table related functions. (2) The presentation assumed TTJ is on tuple-level. However, given the use of pages in real systems, $\mathrm{deleteDT}()$ may be called only after the whole page is processed. Thus, adapting TTJ to work with pages is a future work. (3) TTJ relies on a synchronous assumption that a dangling tuple is removed immediately after its detection. Thus, additional work is needed to make TTJ work without synchronous assumption.

Formal issues. When evaluating acyclic full CQs, TTJ is optimal in data complexity, but is not in combined complexity. In combined complexity, TTJ has an additional $\log k$ term. Yannakakis's algorithm [54] is optimal in both models. A first question is whether there is an improvement to TTJ and/or a better proof that will show that TTJ is optimal in combined complexity as well. Complexity results for TTJ or a TTJ like operator used in bushy plans is also an open issue. There may be mappings between a query's $\mathcal{T}_Q$ and bushy plans such that the size of the materialized output from subplans can be bounded by the final output size. If so, it should follow that that change in mapping definition will show optimality of TTJ for bushy plans as well. It is our conjecture that TTJ can be extended to cyclic queries. That extension will likely require $\mathrm{deleteDT}()$ to pass more information down the plan than it currently does. Any aforementioned work related to practical concerns can also be examined through the formal lens.

ACKNOWLEDGMENTS

Zeyuan Hu would like to thank Yixuan Ni for helpful discussion on various proofs in the paper. Kexin Rong motivates Zeyuan Hu to perform experimental evaluation on TTJ. Discussion with Remmy Yishu Wang helps to clarify acyclicity of JOB queries. We also thank the PODS 2022 reviewers and Hannah K. Amundson for their many insightful comments on earlier drafts of this paper. Zeyuan Hu is supported by University of Texas at Austin teaching assistantship in fall and spring semesters and his own savings in summer.

REFERENCES

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

[2] 2022. Private communication with Philip A. Bernstein.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Vol. 8. Addison-Wesley Reading.

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-Type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3034786.3056105

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *Design and Analysis of Computer Algorithms*. Addison-Wesley.

[6] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2022. *Database Theory*. Open source at https://github.com/pdm-book/community.

[7] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Cambridge, Massachusetts, USA) *(PODS '86)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/6012.15399

[8] Roberto J. Bayardo Jr and Daniel P. Miranker. 1994. An Optimal Backtrack Algorithm for Tree-Structured Constraint Satisfaction Problems. *Artificial Intelligence* 71, 1 (1994), 159–181.

[9] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the Desirability of Acyclic Database Schemes. *J. ACM* 30, 3 (July 1983), 479–513. https://doi.org/10.1145/2402.322389

[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[11] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (Jan. 1981), 25–40. https://doi.org/10.1145/322234.322238

[12] Ming-Syan Chen and Philip S. Yu. 1990. Using Join Operations as Reducers in Distributed Query Processing. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems* (Dublin, Ireland) *(DPDS '90)*. Association for Computing Machinery, New York, NY, USA, 116–123. https://doi.org/10.1145/319057.319074

[13] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Database Theory — ICDT '95*, Georg Gottlob and Moshe Y. Vardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–67.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT press.

[15] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-Aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2011–2026. https://doi.org/10.1145/3318464.3389769

[16] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 418–431. https://doi.org/10.1145/3448016.3457270

[17] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3535 – 3547.

[18] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2nd ed.). Prentice Hall Press, USA.

[19] 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. https://doi.org/10.1145/2902251.2902309

[20] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. 2012. Size and Treewidth Bounds for Conjunctive Queries. *J. ACM* 59, 3, Article 16 (June 2012), 35 pages. https://doi.org/10.1145/2220357.2220363

[21] Danièle Grady and Claude Puech. 1989. On the Effect of Join Operations on Relation Sizes. *ACM Trans. Database Syst.* 14, 4 (Dec. 1989), 574–603. https://doi.org/10.1145/76902.76907

[22] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. https://doi.org/10.1145/152610.152611

[23] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Transactions on Algorithms (TALG)* 11, 1 (2014), 1–20.

[24] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a database system. *Foundations and Trends® in Databases* 1, 2 (2007), 141–259.

[25] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 774–783.

[26] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph-Workloads Through Predefined Joins. *PVLDB 15* (2022).

[27] Stasys Jukna. 2011. *Extremal Combinatorics: With Applications in Computer Science* (2nd ed.). Springer Publishing Company, Incorporated.

[28] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (nov 2019), 252–265. https://doi.org/10.14778/3368289.3368292

[29] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.* 41, 4, Article 22 (Nov. 2016), 45 pages. https://doi.org/10.1145/2967101

[30] 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. https://doi.org/10.14778/2850583.2850594

[31] David Maier. 1983. *The Theory of Relational Databases*. Computer Science Press. http://web.cecs.pdx.edu/%7Emaier/TheoryBook/TRD.html

[32] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of Magic-Sets in a Relational Database System. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/191839.191860

[33] Thomas Neumann and Gerhard Weikum. 2009. Scalable Join Processing on Very Large RDF Graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 627–640. https://doi.org/10.1145/1559845.1559911

[34] 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. https://doi.org/10.1145/3180143

[35] Oracle. [n.d.]. Chapter 4. Types, Values, and Variables. https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.3

[36] Oracle. [n.d.]. Understanding Class Members. https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

[37] Wilson Qin and Stratos Idreos. 2016. Adaptive Data Skipping in Main-Memory Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2255–2256. https://doi.org/10.1145/2882903.2914836

[38] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[39] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). McGraw-Hill.

[40] Kenneth Rosen. 2011. *Discrete Mathematics and Its Applications* (7th ed.). McGraw Hill.

[41] S. Russell and P. Norvig. 2010. *Artificial Intelligence: A Modern Approach* (third ed.). Prentice Hall, Upper Saddle River, NJ. http://aima.cs.berkeley.edu/

[42] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/233269.233360

[43] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[44] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization Strategies in the Vertica Analytic Database: Lessons Learned. In *2013 IEEE 29th International*

Conference on Data Engineering (ICDE). IEEE, 1196–1207.

[45] Abraham Silberschatz, Henry F. Korth, and Shashank Sudarshan. 2019. *Database System Concepts* (7th ed.). McGraw-Hill New York.

[46] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal* 6 (1997), 191–208. https://doi-org.ezproxy.lib.utexas.edu/10.1007/s007780050040

[47] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. 2001. Integrating Semi-Join-Reducers into State-of-the-Art Query Processors. In *Proceedings 17th International Conference on Data Engineering*. 575–584. https://doi.org/10.1109/ICDE.2001.914872

[48] 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.

[49] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. (2020), 2659–2665. https://doi.org/10.1145/3318464.3383132

[50] Allen Van Gelder. 1993. Multiple Join Size Estimation by Virtual Domains (Extended Abstract). In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Washington, D.C., USA) *(PODS '93)*. Association for Computing Machinery, New York, NY, USA, 180–189. https://doi.org/10.1145/153850.153872

[51] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (San Francisco, California, USA) *(STOC '82)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/800070.802186

[52] Todd L Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. *ICDT* (2014). https://doi.org/10.5441/002/icdt.2014.13

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

[54] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*, Vol. 81. 82–94.

[55] C. T. Yu and C. C. Chang. 1984. Distributed Query Processing. *ACM Comput. Surv.* 16, 4 (Dec. 1984), 399–433. https://doi.org/10.1145/3872.3874

[56] Clement T. Yu, Z. Meral Ozsoyoglu, and K. Lam. 1984. Optimization of Distributed Tree Queries. *J. Comput. System Sci.* 29, 3 (1984), 409–445. https://doi.org/10.1016/0022-0000(84)90007-2

[57] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. *Proc. VLDB Endow.* 10, 8 (April 2017), 889–900. https://doi.org/10.14778/3090163.3090167