TreeTracker Join: A Composable Join Algorithm that Yields Optimal Acyclic Multi-Way Joins

Anonymous Author(s)

ABSTRACT

Improving the speed of a relational join is of constant interest. In database theory, continual refinements of applicable algorithmic complexity models serve to focus attention on different fundamentals of the computation and have led to new optimal algorithms. Yet these formal algorithmic improvements rarely make their way into fielded general purpose relational query systems.

TreeTracker Join (TTJ) is a join algorithm that enables the optimal execution of acyclic conjunctive queries and that embodies a solution to two impediments to the practical deployment of optimal join algorithms. First, unlike k-way optimal join algorithms that have k inputs, TTJ takes two relations as input and produces a third relation as output making it compatible with traditional relational query systems. Only upon considering a query plan composed of $k-1$ instances of TTJ can one determine that the ensemble computes the result of an acyclic k-way join in $O(n+r)$ data complexity, where n and r are the input and output sizes. This matches the optimal bound first established by Yannakakis's algorithm. Second, TTJ accomplishes this without introducing semi-join operators. Introducing semi-join operators enlarges the query plan and commonly results in a net reduction of execution speed despite improving the algorithmic complexity.

CCS CONCEPTS

• **Information systems → Join algorithms**.

KEYWORDS

optimal join algorithm, join operator, acyclic conjunctive queries

ACM Reference Format:

Anonymous Author(s). 2021. TreeTracker Join: A Composable Join Algorithm that Yields Optimal Acyclic Multi-Way Joins. In *PODS '22: The Principles of Database Systems (PODS), June 12–17, 2022, Philadelphia, PA.* ACM, New York, NY, USA, 10 pages. https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

1 INTRODUCTION

Improving join performance is of ongoing interest to the entire database community. In database theory, forward progress with respect to formal algorithmic measures has been continually made but, typically, only in association with refinements of the optimality condition. Yannakakis [54] was the first to show that acyclic conjunctive queries can be evaluated optimally with respect to input and output size. More recently, Ngo *et al.* [39] proved that the same bound is unattainable for cyclic queries under tuple-based binary join query plan. This result is motivated by a bound proposed by Atserias *et al.* [6] on the worst-case output size of a k-way join. Subsequently, Ngo *et al.* [39] proposed a new optimality measure with respect to input and worst-case output size. A new class of optimal join algorithms followed [37, 39, 52] coined *worst-case optimal join algorithms (WCOJAs)*. Further advances concern output-sensitive join algorithms [4, 20, 41] and algorithms with stronger optimality [32, 38].

Despite these algorithmic advances, the techniques struggle with respect to their integration with, and impact on, existing relational query systems. For example, Yannakakis's Algorithm plays a central role in hypertree decomposition based join algorithms [4, 20] and related systems [1, 31]. However, to implement Yannakakis's Algorithm in an actual query system, it is necessary to introduce semi-join operators in the query plan. Extensive study [13, 47, 55, 56] has been done on optimizing queries by introducing semi-join operators. However, as noted by Stocker *et al.* [47], introducing semi-joins into query optimization increases plan search space dramatically and, quite often, the goal of removing dangling tuples clashes with finding good plans. In addition, the introduction of semi-join reduction complicates intermediate result size estimation [23] and even instigates faulty results [50].

Another practical challenge appears when optimal algorithms are captured by special join operators. For example, WCOJAs are captured as multi-way join operators [1, 5, 17, 31, 36, 53]. Such multi-way join operators take k inputs to compute a k-way join. Query systems must then represent, and optimize query plans containing traditional unary and binary operators then add k-ary operators. At least one effort determined it was best to completely abandon the relational algebra approach and start from scratch [1]. Even if that direction proves fruitful, existing systems are unlikely to re-engineer such a large and important aspect of their systems.

These observations provided criteria for the design of *TreeTracker Join (TTJ)*, a join algorithm that enables the execution of acyclic conjunctive queries with the same bound as Yannakakis's algorithm and avoids two impediments that limit the use of optimal algorithms in practice.

We suggest two algorithmic elements that are *practical constraints* and must be attained to achieve integration with current RDBMSs:

(1) the signature of the algorithm (the data type of the inputs and output) must be consistent with the traditional signature(s) used to implement binary relational operators. Thus, the operators can be composed with other relational operators and represented in conventional query plans.
(2) the algorithm must avoid a multi-phase algorithm structure. The problematic elements covered by this constraint include

PODS '22, June 12–17, 2022, Philadelphia, PA

ACM ISBN XXX-X-XXXX-XXXX-X/XX/XX...$15.00
https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

increasing the number of operators in a query plan and increasing the number of I/O passes on the base relations. These introduce real costs that are often omitted in algorithmic complexity models.

TTJ is not the first optimal join algorithm result with a practical focus. However, the others do not fulfill the aforementioned practical constraints. Pagh and Pagh [42] developed an I/O efficient Yannakakis's algorithm. However, their results retain the full reducer steps of Yannakakis's algorithm and thus do not fulfill constraint (2). Hu and Yi [27] devised worst-case I/O-optimal join algorithms for acyclic queries. Like most WCOJAs, their algorithms take k relations as input and do not fulfill constraint (1). Ciucanu and Olteanu [14] observed this problem and developed ternary join operators to work with a factorized representation [40] of intermediate results. They achieved WCOJA optimality in a join-at-a-time fashion. However, their design violates constraint (1) due to a disruptive change to the standard binary/unary operator interfaces.

In this paper, we introduce TTJ that satisfies these practical constraints while having the same optimality as Yannakakis's algorithm. A key insight is that dangling tuples can be identified and deleted on the fly during query evaluation thereby avoiding preprocessing. As a side effect, the input relations can be incrementally reduced such that at quiescence, their contents sufficiently approximate the results of running reducing semi-join program [11]. As a result, we are able to achieve optimal $O(n + r)$[1] without introducing explicit semi-joins.

Example 1. Consider a simple chain query $S(x, y) \bowtie B(y, z)$. We use Algorithm 1.1 to compute the join result. We assume both S and B are passed to the algorithm by reference.

Algorithm 1.1: Modified Nested-Loop Join to illustrate TTJ idea

Input: two relations $S(x, y)$ and $B(y, z)$
Output: join result *res*

```
1  res ← ∅
2  ng ← ∅
3  for s ∈ S(x, y) do
4      dangle ← true
5      if s ∉ ng then
6          for b ∈ B(y, z) do
7              if (t ← s⋈b) ≠ nil then
8                  dangle ← false
9                  add t to res
10         if dangle = true then
11             ng ← ng ∪ {s}
12 S ← S − ng
13 return res
```

Algorithm 1.1 is an enhanced nested-loop join: once s is identified as a dangling tuple, the algorithm can add s to *no-good list (ng)* such that if a duplicate tuple of s shows up again, its iteration will be skipped. Note $ng = S \bar{\ltimes} B$ when Algorithm 1.1 reaches Line 12 and subsequently, S is semi-join reduced with respect to B.

Concisely stated, Algorithm 1.1, is an augmentation of nested-loop join that simultaneously computes $S \bowtie\!\!\!\ltimes B$ (i.e., two different relational operators are computed by a single algorithm). The remainder of the paper evolves this concept into an operator that satisfies the practical constraints detailed above. Further, we show a composition of $k-1$ TTJ operators computes a k-way acyclic conjunctive queries in $O(n + r)$, which is optimal.

For those readers familiar with *constraint satisfaction problem (CSP)* solving algorithms, we point out that Algorithm 1.1 was derived from the *TreeTracker-2 (TT-2)* Algorithm for a CSP limited to two variables [8]. The aspect detailed in Algorithm 1.1 as identifying and deleting a dangling tuple is called *learning a no-good* (Section 6.3 in [43]) in the CSP literature. Given the equivalence between CSP and query evaluation [33], it is not surprising that, operationally, a relational operator, semi-join, corresponds to a named technique in CSP. Bayardo and Miranker [8] showed that TT-2 can solve tree-structured CSPs optimally and without an explicit preprocessing step. A primary contribution of this paper is the capture of that technique to compute all the results of a join and to do so within the structure of a composable operator.

For pedagogical purposes, the paper develops TTJ in steps. After preliminaries (Section 2), we first prove in Section 3 that limiting preprocessing of an acyclic conjunctive query to the reducing semi-join program [11] is sufficient for an algorithm otherwise identical to Yannakakis's algorithm to be correct and optimal. We then, in Section 4, define a join operator that can be composed with itself and if the inputs of an acyclic conjunctive query have already been preprocessed by a reducing semi-join program, the set of the composed operators will compute a k-way join optimally. Our main contribution, TTJ, in Section 5, removes the preprocessing assumption used in Section 4 by integrating the idea of Algorithm 1.1 into the operator of Section 4, thereby creating a single operator that computes a join and effects the advantages of semi-join preprocessing. The essence is, if the operator is defined as an object, an additional method called `RemoveDanglingT()` is added to the iterator interface. `RemoveDanglingT()` implements the removal of dangling tuples during join computation by sending information down the query plan, which is akin to *Sideway Information Passing (SIP)* and *Magic Sets* (Section 6).

2 PRELIMINARIES

Conjuctive queries (CQs) correspond to select-project-join queries in relational algebra [21]. To simplify the presentation, we discuss only full CQs, which correspond to a natural join of k relations. A subclass of CQs is acyclic CQs. Many different definitions of acyclic CQs have been purposed and shown to be equivalent [2, 9, 34]. Herein, we use the join tree definition of acyclic CQs. A *Join tree*, G_Q, is an acyclic query graph [12] with one additional constraint: for each pair of distinct nodes R_1, R_2 in the tree and for every common attribute a between R_1 and R_2, every relation on the path between R_1 and R_2 contains a [9].

[1] In this paper, the big-O notation is in data complexity ignoring terms that depending on query expression not data, and big-O indicates the combined complexity [51].

Yannakakis's Algorithm [54] evaluates acyclic CQs optimally $O(n + r)$ with input size n and output size r. It requires three-passes over the join tree. The algorithm first runs a *reducing semi-join program* [11] by traversing the join tree bottom-up and applying $R_p \ltimes R_c$ where R_p is a parent relation and R_c is one of its children. We use $P_{Q,i}$ to denote reducing semi-join program on G_Q with root R_i (when we do not need to emphasize that R_i is the root, we simply writes P_Q). The resulting relations after P_Q are denoted R'_i. In the second pass, the algorithm traverses the join tree top-down applying $R'_c \ltimes R'_p$ ($R_c \ltimes R'_p$ if R_c is a leaf node). The fully reduced relations are denoted R^*_i for $i \in [k]$ [2]. The third pass produces the join output by again traversing join tree bottom-up. The *data complexity* [51] of Yannakakis's algorithm is $O(n + r)$ with n being the relation size and r being the output size. The key ingredient in Yannakakis's algorithm is the full reducer's complete removal of *dangling tuples* (i.e., those tuples that do not appear in the final join result).

In practice, a query is translated into a query plan comprising relational algebra operators. Often, query engines are architected as dataflow systems [25, 26]. That architecture is extensible and effectively supports parallel execution. An implementation characteristic of such systems is the physical implementation of the operators using iterators [24]. An iterator is a base class inherited by all of the physical relational operators that includes three methods: `Open()` (initialize internal state and set up dataflow), `GetNext()` (produce an output tuple from some computation), and `Close()` (clean up state) [18, 19]. To evaluate queries, query systems commonly organize operators as a *left-deep query plan* (a binary tree with its all right children base relations [19]) and use a demand-driven pipelining physical plan evaluation strategy [46] shown in Algorithm 2.1 to obtain query result. In this paper, we use the same strategy to drive operators in our algorithms to evaluate queries.

Algorithm 2.1: Driver program to evaluate Q

Input: root i of a left-deep query plan
Output: join result res

$res \leftarrow \emptyset$
i.`Open()`
while ($r \leftarrow i$.`GetNext()`) $\neq$ *nil* **do**
    add r to res
i.`Close()`
return res

2.1 Additional Notation

We denote a database schema as $\boldsymbol{D}$ and a database instance of $\boldsymbol{D}$ as $\boldsymbol{I}$. We consider an acyclic CQ $\boldsymbol{Q}$ of k relations each with size n. Its join tree is G_Q. To evaluate such $\boldsymbol{Q}$, we use pre-order traversal of join tree and place relations in a left-deep query plan in bottom-up fashion - the root of G_Q is the left-most relation at the bottom. For a given join operator in the plan, R_{outer} (R_{inner}) refers to its left (right) child. Join operators in a plan are labeled top-down as $\bowtie_u$ for $u \in [k-1]$ in ascending order. The left-most relation, the root of G_Q, is $\bowtie_k$. $G_{\bowtie_u}$ shall denote the set of relations in the query plan that below $\bowtie_u$. *attr* is a function that extracts attributes from a relation or from each relation in a set of relations and returns their union. In addition, J_u, $u \in [k]$ denotes the join result computed by $\bowtie_u$. J^*_u denotes the join of relations in $G_{\bowtie_u}$. Thus, for a correct join algorithm, $J_u = J^*_u$. Let j_u denote $\bowtie_u$'s result size. In particular, $j_1 = r$, which is the query result size. For a tuple t of $R(a, b)$, we use both a named perspective (e.g., $(a : 1,\ b : 2)$) and an unnamed perspective (e.g., $R(1,\ 2)$) to represent t interchangeably [2]. $t[a] = \pi_a(t)$ for tuple t and attribute a. For tuple t and relations R, S, let join attribute value $jav(t, R, S) = t[attr(R) \cap attr(S)]$. We assume standard RAM complexity model.

[2] $[k]$ is a shorthand for $\{1, \ldots, k\}$ [30].

3 REDUCING SEMI-JOIN PROGRAM IS ENOUGH

Algorithm 1.1 indicates that P_Q can be interwoven with join computation, which implies two-passes over G_Q is sufficient to compute the join result. Algorithm 1.1 is enumerating output top-down over G_Q and interweavingly, doing bottom-up semi-join operations. In other words, there is one redundant pass in Yannakakis's algorithm, which comes from the full reducer, that makes it impractical.

Theorem 3.1. *Given a join tree G_Q and root R_1, one can compute join with the following two steps:*

(1) apply $P_{Q,1}$ on G_Q;
(2) perform pair-wise join from root R_1 to leaves recursively.

Any intermediate join result during the computation will not contain any dangling tuples.

The intuition is that after applying $P_{Q,1}$, $R_1 = R^*_1$ (Lemma 4 of [11]) and each other relation only contains tuples that are joinable with its child relations. If we start to compute join from this state in a top-down fashion, it is impossible to produce dangling tuples. Detailed proof of Theorem 3.1 is in Appendix A. We use Example 2 to illustrate the extra work done by Yannakakis's algorithm.

Example 2. Suppose there are three relations in G_Q: R_p, R_j, and R_i. R_p is the parent of R_j and R_j is the parent of R_i. To evaluate Q using Yannakakis's algorithm, the following operations are carried out:

$$R'_j = R_j \ltimes R_i \quad (1)$$

$$R'_p = R_p \ltimes R'_j \quad (2)$$

$$R^*_j = R'_j \ltimes R'_p \quad (3)$$

$$R^*_i = R_i \ltimes R^*_j \quad (4)$$

Theorem 3.1 executes (1) and (2). Join is executed starting at R'_p. For $R'_p \bowtie R'_j$, tuples in $R'_j \,\bar{\ltimes}\, R'_p$ will not be selected. Thus, (3) is not needed. Similarly, (4) is not needed. The reason that Yannakakis's algorithm requires (3) and (4) is because the join is performed in a bottom-up fashion. Without first removing dangling tuples in R_j and R_i, Yannakakis's algorithm may produce an unjoinable intermediate result. Thus, the additional semi-joins are needed to achieve the complexity bound but not the correctness of the algorithm.

Corollary 3.2. *The algorithm in Theorem 3.1 runs $O(n + r)$, which is the same as Yannakakis's algorithm.*

Corollary 3.2 immediately follows from Theorem 3.1 because intermediate result size is smaller than the final result size.

Empirically and without proof of correctness of the system, EmptyHeaded [1] (Section 3.5) eliminates the top-down pass of Yannakakis's algorithm and demonstrates a 10% performance improvement for tested workload.

4 TREETRACKER-γ JOIN

We first define the *TreeTracker-γ (TT-γ)* Join (Algorithm 4.1, 4.2, and 4.3) that assumes step (1) of Theorem 3.1 is done and computes step (2) of Theorem 3.1 in Yannakakis bound. TT-γ forms the basis of TTJ and plays an important role in TTJ runtime analysis.

Algorithm 4.1: Open() of Join Operator (TT-γ & TTJ)

Global variables: $r_{inner}, r_{outer}, R_{inner}, R_{outer}, l, I_l$
Output: One joined tuple of R_{outer}, R_{inner} (i.e., one row of $R_{outer} \bowtie R_{inner}$)

Function `Open()`:
  $l \leftarrow nil$
  $I_l \leftarrow nil$
  $r_{inner} \leftarrow nil$
  $r_{outer} \leftarrow nil$
  $H \leftarrow$ empty hash table
  R_{inner}.`Open()`
  while ($r_{inner} \leftarrow R_{inner}$.`GetNext()`) $\neq nil$ **do**
    $H[jav] \leftarrow H[jav] \cup \{r_{inner}\}$ where $jav = jav(r_{inner}, R_{inner}, R_{outer})$
  R_{outer}.`Open()`

Algorithm 4.2: GetNext() of Join Operator in TT-γ

1 **Function** `GetNext()`:
2   **if** $l \neq nil$ **then**
3   advance I_l
4   **if** $I_l \neq nil$ **then**
5   **return** join of element pointed by I_l with r_{outer}
6   $r_{outer} \leftarrow R_{outer}$.`GetNext()`
7   **if** $r_{outer} = nil$ **then**
8   **return** *nil*
9   **return** `LookUpH()`

Here are a few remarks on the algorithm details:

- Consider Algorithm 4.1, 4.2, and 4.3 to be methods associated with operators. Thus, global variables first listed in Algorithm 4.1 are within the scope of all three algorithms. Methods associated with the operator can change the state of those variables during the runtime. Further, those variables are not accessible by other operator instances.

Algorithm 4.3: LookUpH() of Join Operator in TT-γ

Function `LookUpH()`:
  $l \leftarrow H[jav]$ with jav computed from r_{outer}
  if $l \neq nil$ **then**
    initialize I_l pointing to the first element of l
    return join of the element pointed by I_l with r_{outer}
  return *nil*

- H is a hash table with jav computed from r_{inner} as its key and its value is a list of tuples from R_{inner} sharing the same jav.
- `LoopUpH()` is a private method that is invoked by `GetNext()`. Thus, no modification is made to the iterator interface.

TT-γ join algorithm is very similar to hash join (Table 1 in [24]). The only difference between TT-γ and hash join happens inside `GetNext()` starting at Line 6. Since P_Q is already applied and relations are ordered in pre-order traversal, by Theorem 3.1, any non-nil r_{outer} returned from Line 6 is joinable. Thus, algorithm can call `LookUpH()` to compute the join result.

Theorem 4.1. *TT-γ Join Algorithm (Algorithm 4.1, 4.2, and 4.3) computes the correct join result.*

Proof. Proof by induction on the join operator u. Base case, $u = k$. Because $\bowtie_k$ is the root of G_Q and P_Q has been applied, the claim holds following Theorem 3.1. Assume the claim holds for $u = i$ (i.e., $J_i = J_i^*$), we want to show it holds for $u = i - 1$. Let r_{outer}^j denote the jth value assigned to r_{outer}. `LookUpH()` is called for each new r_{outer} from $\bowtie_i$. Thus, for each non-nil r_{outer}^j with $j \in [j_i]$ from J_i, $l = R_{i-1} \ltimes \left\{r_{outer}^j\right\}$. Since I_l is never reset until $\left\{r_{outer}^j\right\} \bowtie l$ is computed, Thus, tuples returned by $\bowtie_{i-1}$ equals to

$$\bigcup_{j=1}^{j_i} (R_{i-1} \ltimes \left\{r_{outer}^j\right\}) \bowtie \left\{r_{outer}^j\right\}$$

, which is J_{i-1}^*. □

Theorem 4.2. *The runtime complexity of evaluating Q assuming application of P_Q using TT-γ Join Algorithm (Algorithm 4.1, 4.2, 4.3) driven by Algorithm 2.1 is $O(n + r)$.*

Proof. 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 H. By Theorem 3.1, 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 $O(kn + kr) = O(n + r)$. □

5 TREETRACKER JOIN

To define TTJ (Algorithms 4.1, 5.1, 5.2, 5.3, 5.4, 5.5, and 5.6), we integrate the removal of dangling tuples into the TT-γ algorithm, thereby eliminating the preprocessing reduction step assumed to have occurred in the previous section.

Intuitively, to eliminate the explicit preprocessing step, we are integrating the concept first shown in Algorithm 1.1. The proper algorithmic changes are actually accomplished by interweaving step (1) and step (2) of Theorem 3.1. Yet, like Algorithm 1.1, TTJ takes a fine-grained approach. Instead of doing join and semi-join on sets of tuples, the same results are achieved in a tuple-by-tuple fashion. The number of dangling tuples removed by TTJ is no greater than the number removed by P_Q[3] and they may be removed prior to the completion of TTJ's execution. If so, at that point, TTJ will have the same behavior as TT-γ. TTJ takes no more than the time P_Q takes to remove dangling tuples, $O(n)$. Since TT-γ meets Yannakakis's algorithm optimality, TTJ also achieves the desired bound.

Algorithm 5.1: GetNext() of Join Operator in TTJ

1 **Function** `GetNext()`:
2   **return** `GetOuter(`$getNewOuterTuple = true$`)`

Algorithm 5.2: LookUpH() of Join Operator in TTJ

1 **Function** `LookUpH()`:
2   **if** $l = nil$ **then**
3         $l \leftarrow H[jav]$ with $jav = jav(r_{outer}, R_{inner}, R_{outer})$
4   **if** $l \neq nil$ **then**
5   initialize I_l to point to the first element of l
6   **return** join of the element pointed by I_l with r_{outer}
7   **else**
8   advance I_l pointing to the next element of l
9   **return** the join of element pointed by I_l with r_{outer}
10   **return** nil

Example 3. Let $\boldsymbol{D} = \{T(x),\ S(x,y,z),\ B(z),\ R(y,z)\}$, $\boldsymbol{I} = \{T(green),\ T(red),\ S(red,1,2),\ S(red,3,2),\ B(2),\ R(3,2)\}$, and $G_Q = \{(T,S),\ (S,B),\ (S,R)\}$ in edge list representation with root T. $\boldsymbol{Q}$ over $\boldsymbol{D}$ has exactly one result: $(x: red,\ y: 3,\ z: 2)$. Starting with the driver (Algorithm 2.1), `GetNext()` makes recursive calls to itself ending with a call to T's table scan operator (Algorithm 5.6). The table scan operator's ng value is empty, so $T(green)$ is returned (indicated by $\rightarrow$ in Figure 1 **(A)**). `LookUpH()` is called by operator instance $\bowtie_3$ (Algorithm 5.3 Line 12). Figure 1 **(A)** illustrates the resulting state.

Since none of the tuples in S can join with $T(green)$, nil is returned (Algorithm 5.2 Line 10). $\bowtie_3$ calls `RemoveDanglingT()` (Algorithm 5.3 Line 15). For $\bowtie_3$, R_{outer} references T and R_{inner} references S. Thus, T.`RemoveDanglingT(S)` is called (indicated by $\leftarrow$ in Figure 1 **(B)**). The call to the table scan operator's `RemoveDanglingT()` (Algorithm 5.5) results in $T(green)$ being added to ng. The next tuple that is not in ng, $T(red)$, is returned. Figure 1 **(B)** illustrates this state.

Operator $\bowtie_3$'s r_{outer} input contains $T(red)$. `LookUpH()` is called by $\bowtie_3$ from Algorithm 5.3 Line 12. A lookup on hash table H, which contains S, returns $\{(x: red,\ y: 1,\ z: 2)\}$ as l (Algorithm 5.2 Line 3). `LookUpH()` in $\bowtie_3$ per Algorithm 5.3 Line 14 returns $(x: red,\ y: 1,\ z: 2)$. The value of r_{outer} in operator instance $\bowtie_2$ is set to $(x: red,\ y: 1,\ z: 2)$ by Algorithm 5.3 Line 7. The call to `LookUpH()`

[3]See Lemma 5.1 and Corollary 5.2.

Algorithm 5.3: GetOuter() of Join Operator in TTJ

1 **Function** `GetOuter(`$getNewOuterTuple$`)`:
2   **if** $getNewOuterTuple = true$ **then**
3   **if** $l \neq nil$ **then**
4   advance I_l
5   **if** $I_l \neq nil$ **then**
6   **return** join of element pointed by I_l with r_{outer}
7         $r_{outer} \leftarrow R_{outer}$.`GetNext()`
8         $l \leftarrow nil$
9   **if** $r_{outer} = nil$ **then**
10   **return** nil
11   **while** $true$ **do**
12         $r_{inner} \leftarrow$ `LookUpH()`
13   **if** $r_{inner} \neq nil$ **then**
14   **return** r_{inner}
15         $r_{outer} \leftarrow R_{outer}$.`RemoveDanglingT(`$R_{inner}$`)`
16   **if** $r_{outer} = nil$ **then**
17   **return** nil
18         $l \leftarrow nil$

Algorithm 5.4: RemoveDanglingT() of Join Operator in TTJ

1 **Function** `RemoveDanglingT(`$\boldsymbol{relation}$`)`:
2   **if** R_{inner} *is the parent of* $\boldsymbol{relation}$ *in* G_Q **then**
3   Remove tuple pointed by I_l
4   **if** H *is empty* **then**
5   **return** nil
6   **else**
7         $r_{outer} \leftarrow R_{outer}$.`RemoveDanglingT(`$\boldsymbol{relation}$`)`
8         $l \leftarrow nil$
9   **return** `GetOuter(`$getNewOuterTuple = false$`)`

Algorithm 5.5: RemoveDanglingT() of Table Scan Operator in TTJ

1 **Function** `RemoveDanglingT(`$\boldsymbol{relation}$`)`:
    `// ng is a set of tuples.`
2   put the tuple last returned in ng
3   **return** `GetNext()`

Algorithm 5.6: GetNext() of Table Scan Operator in TTJ

1 **Function** `GetNext()`:
2   **return** the next tuple that is not in ng

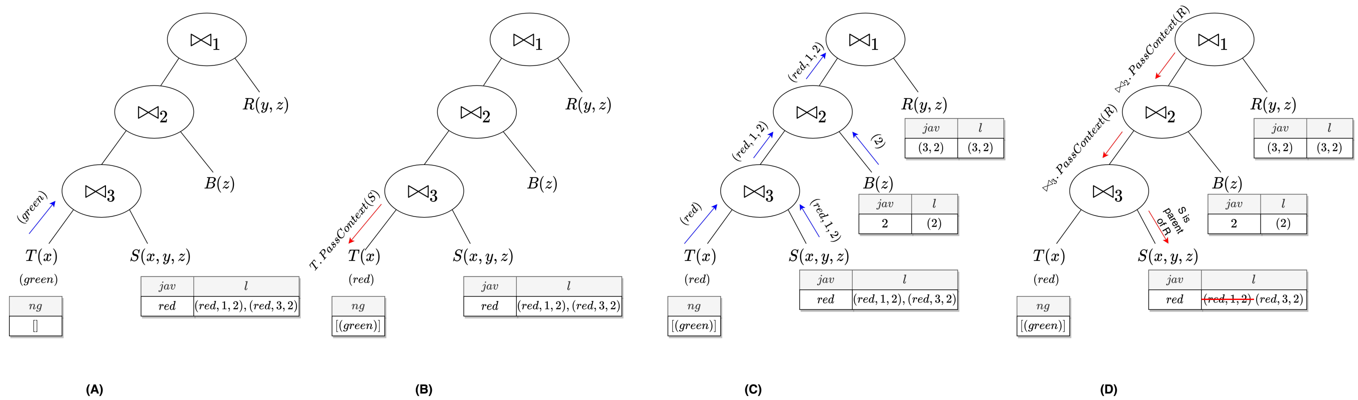

Figure 1: The figure shows four execution states of TTJ when evaluating $T(x)\bowtie S(x,y,z)\bowtie B(z)\bowtie R(y,z)$ over I in Example 3.

returns $(x:red,\ y:1,\ z:2)$ and set as $\bowtie_1$'s r_{outer} value. Operator $\bowtie_1$ then calls `LookUpH()`. Figure 1 **(C)** illustrates this state.

Looking up R tuples in H of $\bowtie_1$ returns nothing because $y = 1$ in tuple $(x:red,\ y:1,\ z:2)$ fails to join. Thus, operator $\bowtie_1$ calls `RemoveDanglingT()` (Algorithm 5.3 Line 15) with argument R. R_{outer} is referencing $\bowtie_2$. Since B is not the parent of R in G_Q, `RemoveDanglingT()` is recursively called from Algorithm 5.4 Line 7 with R_{outer} references $\bowtie_3$. S is the parent of R in G_Q. The algorithm removes $S(red,\ 1,\ 2)$, which is pointed by I_l (Algorithm 5.4 Line 3). Figure 1 **(D)** illustrates that state.

Looking into H of R in $\bowtie_1$ returns nothing because $y = 1$ from $(x:red,\ y:1,\ z:2)$ fails the join. $\bowtie_1$ calls `RemoveDanglingT()` (Algorithm 5.3 Line 15) with argument R. R_{outer} in $\bowtie_1$ references $\bowtie_2$. Since B is not the parent of R in G_Q, `RemoveDanglingT()` is recursively called from Algorithm 5.4 Line 7 with R_{outer} references $\bowtie_3$. S is the parent of R in G_Q. The algorithm removes $S(red,\ 1,\ 2)$, which is pointed by I_l (Algorithm 5.4 Line 3). Figure 1 **(D)** illustrates the state of operators at this moment.

The algorithm calls `GetOuter(false)` from Algorithm 5.4 Line 9 so that H of S can be checked again to see if there is another tuple joining with $T(red)$. In this case, $(red,\ 3,\ 2)$ does and the join result is computed.

TTJ has similar structure as TT-γ but adds the method `RemoveDanglingT()`. Technically, `RemoveDanglingT()` is a third input to the operator. However it is strictly additive to existing interfaces, does not need to be implemented by other operators and thus, as a practical matter, does not pose a challenge to constraint (1).

TTJ implements the concept presented as Algorithm 1.1 but does so in the form of a composable operator. Algorithm 1.1 achieves semi-join reduction by removing dangling tuples from a base relation, which is not possible if the algorithm is embedded in a relational query system. To achieve the same effect, TTJ, like a hash join, reads one of its relational arguments and initializes a local hash index, H_i, per the contents of the relation R_i for $i \in [k-1]$. As dangling tuples are identified, they can be removed from further consideration by removing them from H_i, limiting the scope of the side effect to inside the operator. Similar mechanism for R_k is a deny list *ng*, which works the same as shown in Algorithm 1.1.

In Example 1, S is the parent of R. In Algorithm 1.1, once $s \in S$ is detected as a dangling tuple, the execution flow can switch from inner loop associated with B to outer loop associated with S and modify its *ng* value. However, in query plan, this mechanism is not built-in. Thus, `RemoveDanglingT()` is needed to change evaluation execution flow; just like `GoTo` in programming languages. When a dangling tuple is detected by R_i, the execution should directly jump back to R_i's parent, R_j, and remove R_j's tuple pointed by I_l because by G_Q definition, R_j is the source of the failure. Thus, `RemoveDanglingT()` is invoked with argument R_i and execution flow restarts from R_j. This disruption with respect to flow of control skips executing unnecessary operations in the operators skipped. This idea is the same as *backjumping* in CSP [43]. Broadly speaking, information of joined tuple flows up in the query plan whereas `RemoveDanglingT()` sends a dangling tuple signal down.

Lemma 5.1 speaks to how TTJ reflects step (1) of Theorem 3.1 in its join computation.

Lemma 5.1. *W.L.O.G, let R_k be the root of G_Q with k relations. Let H'_i with $i \in [k-1]$ denote the initial contents of H_i minus the entries removed by Line 3 in* `RemoveDanglingT()` *(Algorithm 5.4) after evaluating Q with TTJ. We have two families of sets:*

(1) $\mathcal{A} = \{R_k - ng, H'_{k-1}, \dots, H'_1\}$
(2) $\mathcal{B} = \{R'_k, R'_{k-1}, \dots, R'_1\}$ after running $P_{Q,k}$ on G_Q

Then, $R_k - ng = R'_k$ and $R'_i \subseteq H'_i$ for $i \in [k-1]$.

Proof. For each R_i for $i \in [k]$, Denote the set from $\mathcal{A}$ that built from R_i as R_i^A (e.g., $R_1^A = H'_1$ and $R_k^A = R_j - ng$). Similarly, the set from $\mathcal{B}$ denoted as R_i^B. We first show $R_i^B \subseteq R_i^A$.

Case 1. R_i is a leaf node of G_Q. By the definition of P_Q, $R_i^B = R'_i = R_i$. On the other hand, $R_i^A = H'_i = H_i = R_i$ because H_i contains all tuples of R_i and is modified only when R_i is the parent of some node in G_Q. Thus, $R_i^A = R_i^B$ and lemma holds for leaf nodes.

Case 2. R_i is a non-leaf node of G_Q. First consider $i \in [k-1]$, $R_i^A = H'_i$. Suppose $t \notin R_i^A$. This means t is one of the tuples removed by Algorithm 5.4 Line 3. Line 3 is executed only when an intermediate join result, a concatenation of tuples including t, cannot join with one of its child relation R_j in the upper part of plan. Thus, $t \notin R_i^B$ because t cannot join with any tuples in R_j and will be removed by $R_i \ltimes R_j$ in $P_{Q,k}$. Since $t \notin R_i^A$ implies $t \notin R_i^B$, $R_i^B \subseteq R_i^A$ for $i \in [k-1]$. For $i = k$, we have $R_i^A \subseteq R_k - ng$. Suppose $t \notin R_k - ng$. Since ng contains the tuples of R_k that are removed by `RemoveDanglingT()`, for the same reason as above, t cannot join with one of R_k's child. Thus, $t \notin R_k^B$. Thus, $R_i^B \subseteq R_i^A$ for $i \in [k]$.

Implied by Theorem 3.1, it can be the case that $t \in R_i^A$ and $t \notin R_i^B$ for $i \in [k-1]$. Specifically, tuples from R_i that cannot join with any tuples from its parent will not be removed by TTJ. However, some of them can be removed by $P_{Q,k}$ if those tuples cannot join with one of R_i's children. For example, consider $\boldsymbol{D} = \{R_3(x), R_2(x,y), R_1(y)\}$ with $\boldsymbol{I} = \{R_3(4), R_2(4,6), R_2(3,5), R_2(4,7), R_1(7)\}$. Suppose $R_3 \rightarrow R_2 \rightarrow R_1$ is the G_Q. Then, $R_2(3,5)$ will not be removed after TTJ but will be by $P_{Q,k}$. Thus, $R_2(3,5) \in H_2'$ but $R_2(3,5) \notin R_2'$. On the other hand, if tuples from R_i that cannot join with R_i's parent but can join with R_i's children, then $R_i^A \subseteq R_i^B$.

It remains to show $R_k^A \subseteq R_k^B$. Suppose $t \notin R_k'$. t is removed because it cannot join with any tuples from R_j, a R_k's child. $t \notin R_k - ng$. Every tuple of R_k will be returned if it doesn't belong to ng. Then t will be returned. Since none of R_j can join with t, `RemoveDanglingT()` is called. Since R_k is the parent of R_j, t is put onto ng. Since $t \notin R_k'$ implies $t \notin R_k - ng$, $R_k^A \subseteq R_k^B$. Since $R_k^A \supseteq R_k^B$, $R_k^A = R_k^B$. Combining all the cases, the lemma holds under bag semantics. □

Corollary 5.2. *If we measure work, W, done by an algorithm as the number of tuples removed from relations in G_Q, $W^{TTJ} \leq W^{P_Q}$.*

Corollary 5.2 immediately follows from Lemma 5.1. Intuitively, P_Q does redundant work. Reusing Example 2, tuples from $R_j \bar{\ltimes} R_p$ will not fail Theorem 3.1 but some may be removed by P_Q because they cannot join with R_i.

5.1 Correctness of TTJ

Lemma 5.3. *For every assignment to r_{outer}, l is initialized with values in `LookUpH()` and I_l is reset. Between each pair of assignments to r_{outer}, l is never initialized and I_l is never reset.*

Proof. Whenever r_{outer} is assigned, l is set to nil. Since l is initialized and I_l is reset when $l = nil$ in `LookUpH()`, the result follows. □

Theorem 5.4. *TTJ (Algorithms 4.1, 5.1, 5.2, 5.3, and 5.4, 5.5, 5.6) driven by Algorithm 2.1 computes the correct join result .*

Proof. We need to show $J_1 = J_1^*$ with

$$J_1^* = \{t \text{ over } attr(G_{\bowtie_1}) \mid t[attr(R_u)] \in R_u \; \forall u \in [k]\}$$

under bag semantics. We first show $J_1 \subseteq J_1^*$. Let $t \notin J_1^*$. There are two cases.

Case 1. There exists R_i such that $t[attr(R_i)] \notin R_i$. In this case, it is trivial to see that $t \notin J_1$.

Case 2. t satisfies: $\exists R_i$ such that $t[attr(R_i)] = t_i \in R_i$ but $t_i \in R_i \bar{\ltimes} R_j$ for some R_j. We need to show any t satisfying above condition cannot be in J_1. By G_Q definition, relations on the path between R_i and R_j have attributes $attr(R_i) \cap attr(R_j)$. Thus, t also satisfies: $\exists R_x$ such that $t[attr(R_x)] = t_x \in R_x$ but $t_x \in R_x \bar{\ltimes} R_p$ for some relations R_x and R_p on the path between R_i and R_j. Further, R_x and R_p form parent-child relation and are connected by an edge in G_Q. If R_p is the parent and R_x is the child, $t \notin J_1$. Suppose R_p is the child and R_x is the parent. TTJ will call `RemoveDanglingT()` from the join operator connected with R_p and t_x will be deleted from H_x. Thus, t will not be returned and is not in J_1. Note the same execution applies if t values are duplicated. Thus, the condition is satisfied under both set and bag semantics.

To show $J_1^* \subseteq J_1$, suppose $t \in J_u^*$ but $\notin J_u$ for some $u \in [k]$. Since $t \in J_u^*$, $t[attr(R_u)]$ can join with all relations from $u-1$ to 1 in the plan. Thus, $t[attr(R_u)] \in H_u'$. Thus, it must be that $t[attr(G_{\bowtie_{u+1}})] \in J_{u+1}^*$ but $t \notin J_{u+1}$. The same argument applies to every operator in the plan. Eventually, we have $t[attr(R_k)] \in J_k^*$ but $t \notin J_k$. However, this is a contradiction. $t[attr(R_k)] \in J_k^*$ and joins with the rest of the relations in plan. Thus, $t[attr(R_k)] \notin ng$ and $\in J_k$. Since u is picked arbitrarily, $J_1^* \subseteq J_1$.

For $t \in J_1^*$, we need to show the number of tuples t that are in J_1^* equals to the number of tuples t shown in J_1. This follows from Lemma 5.3. The proof similar to Theorem 4.1's proof. □

5.2 Runtime Analysis of TTJ

Definition 1 (clean state). The execution of a query plan reaches a *clean state* if ng and H_u for $u \in [k-1]$ are the same as $\mathcal{A}$ in Lemma 5.1.

The moment after the query execution reaches a clean state, TTJ satisfies Lemma 5.5 and 5.6. The proofs are in Appendix B and Appendix C, respectively.

Lemma 5.5. *$J_u^* \bowtie H_{u-1}'$ will not create dangling tuples.*

Lemma 5.6. *The tuple produced by $\bowtie_u$ will be an element in J_u^* for all $u \in [k]$.*

Theorem 5.7. *The data complexity of evaluating Q using TreeTracker Join Algorithm (Algorithm 4.1, 5.1, 5.2, 5.3, and 5.4, 5.5) driven by Algorithm 2.1 is $O(n+r)$.*

Proof. By Lemma 5.1, the execution of a plan is in clean state when TTJ execution finishes. Thus, the amount of work caused by backtracking via `RemoveDanglingT()` is fixed. Suppose the execution is in clean state after computing the first join result.

We first bound the cost of getting the first join result. `Open()` is $O(kn)$. The total cost of `GetNext()` without counting `RemoveDanglingT()` is bounded by the total number of loops (starting at Line 11) within `GetOuter()` no matter the argument. Each time `RemoveDanglingT()` is called (Algorithm 5.3 Line 15), exactly one tuple is removed from H: since TTJ never re-reads a base relation after H is built, removing an element from H is effectively the same as removing a tuple from the base relation. There can be at most $(k-1)n$ backtracks because once H is empty, `RemoveDanglingT()` returns nil. In addition, for the $\bowtie_1$ operator, the number of `RemoveDanglingT()` calls from Algorithm 5.3 Line 15 is $j_2 + 1$ and for the $\bowtie_2$ operator, the number is $j_3 + 1$, and so on. Thus, $\sum_{i=1}^{k-1}(j_{i+1}+1) = (k-1)n$. In other words, the total number of loops in `GetOuter()` calls is $O((k-1)n)$. Since the number of loops in `GetOuter(true)` and the number of loops in `GetOuter(false)` in total is $O((k-1)n)$, the total cost of `GetNext()` without considering `RemoveDanglingT()` is $O(kn)$.

Next, we bound the cost of `LookUpH()`. Each call takes $O(1)$. Since the total number of `LookUpH()` calls is bounded by the total number of loops in `GetOuter()` no matter the argument, the total cost of `LookUpH()` is $O(kn)$.

Next, we need to count the total number of `RemoveDanglingT()` calls: not just calls from Algorithm 5.3 Line 15 (in total, $O(kn)$) but also the recursive calls made by `RemoveDanglingT()` itself at Algorithm 5.4 Line 7. A call to `RemoveDanglingT()` made in ith operator from Line 15, `RemoveDanglingT()` can be recursively called at most $k - i$ times from Algorithm 5.4 Line 7 and $j_{i+1} + 1$ more calls made in Algorithm 5.3 Line 15 due to additional `GetR1(false)` calls from `RemoveDanglingT()`. Since each relation can be backtracked at most n times, the number of `RemoveDanglingT()` calls with $k - 1$ recursive calls is at most n. The same applies to `RemoveDanglingT()` calls with $k-2, k-3, \ldots, 1$ recursive calls. Thus, the total number of `RemoveDanglingT()` calls is $\sum_{i=1}^{k-1}(k - i) \cdot n + j_{i+1} + 1 = O(k^2 n)$.

For each `RemoveDanglingT()` call, `GetOuter(false)` is called exactly once. Between two `GetOuter(false)` calls, $O(1)$ work is done. Therefore, total amount of work done by `GetOuter(false)` is $O(k^2 n)$.

Summing everything together, it takes $O(k^2 n)$ to compute the first join result. From Lemma 5.5 and Lemma 5.6, once execution reaches a clean state and $J_u \subseteq J_u^*$, there is no backtracking. Thus, there will be no more `RemoveDanglingT()` calls and the result of `LookUpH()` can be returned directly. Thus, once the execution is in a clean state, TTJ behaves exactly the same as TT-γ (Section 4). Since the execution is in a clean state after the first join result is computed, the total cost for computing the r join result is $O(k^2 n + (r-1)k) = O(n + r)$, which is equal to that of Yannakakis's algorithm. □

Corollary 5.8. *TTJ and Yannakakis's algorithm is equivalent from both scope of applicability and algorithmic complexity.*

6 DISCUSSION AND RELATED WORK

From database perspective, `RemoveDanglingT()` is reminiscent of *Sideways Information Passing (SIP)* [7, 10, 28, 45, 57] and *Magic Sets* [7, 10, 35, 44]. TTJ, SIP, and Magic Sets share the same goal of filtering out dangling tuples as early as possible in the query plan. SIP and Magic Sets achieve the goal by sending partial results computed from subpart of the query to the other subpart. TTJ is different from their approach because TTJ never waits for partial results computed before calling `RemoveDanglingT()`; once a dangling tuple is identified, information is sent immediately. In addition, TTJ does not transform query and associated plans; what information to pass is determined at runtime instead of optimization step. However, TTJ is compatible with many existing SIP approaches. For example, Ives and Taylor [28] create a Bloom filter on a computation-completed subtree of a bushy plan and sends the filter to the other subtree to semi-join reduce arriving tuples. TTJ can be directly employed in the subtree computation.

A CSP technique, (hyper)tree decomposition [15, 16, 22], has been successfully adapted and applied in the context of query evaluation [3, 20, 21, 29, 48]. Join algorithms based on hypertree decomposition handle CQs with complexity form $O(n^d + r)$ where d is a *width* parameter determined by the topology of query structure [20]. Tziavelis *et al.* [49] note that those algorithms share the same algorithmic structure and Yannakakis's algorithm as the final step is used to compute the join result on derived relations from the decomposition. Given the equivalence between Yannakakis's algorithm and TTJ, TTJ can directly replace Yannakakis's algorithm to evaluate cyclic CQs with hypertree decomposition.

Note that TTJ cannot be directly applied to cyclic CQs because the join failure may be caused by a combination of values of multiple attributes from different relations. Thus, removing a tuple from a relation that contributes only partial of the combination will lead to incorrect join result. However, TTJ demonstrates that one operator can pass information to another operator with method calls subject to parent-child relation in G_Q. In addition, the dangling tuple information is either explicit or implicit maintained in each operator. It is natural to ask whether it is possible to maintain no-good combination of attribute values in proper operator(s) to achieve reasonable bound for evaluating cyclic CQs. We treat this exploration as part of future work.

7 CONCLUSION AND FUTURE WORK

Being an optimal algorithm for acyclic CQs, Yannakakis's algorithm is hard to use in practice due to additional semi-joins introduced in the full reducer preprocessing step. In this paper, we show that preprocessing relations are not needed to reach optimal evaluation of acyclic CQs. We develop TTJ, a composable join algorithm that has the same bound as the Yannakakis's algorithm. TTJ takes traditional unary and binary operator forms and can be directly used in existing query plans without introducing any extra operators. The key ingredient is, with techniques from CSP, TTJ removes dangling tuples on the fly during join computation. The implication is that a physical operator can implement two relational algebra operations at the same time. Thus, as a future work, it is worth to explore the possibility of mix and match operators shown in Algorithm 1.1 with existing operators to improve overall query performance. In addition, TTJ implements learning no-good idea with the help from object-oriented design pattern: an operator has private fields that can be changed by a side effect of a method call at runtime. Thus, it is interesting to see whether such idea enables the design of practical algorithms that may be seemingly impossible from relational algebra perspective.

REFERENCES

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

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

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. 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, 13–28. https://doi.org/10.1145/2902251.2902280

[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] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD*

International Conference on Management of Data. 1371–1382.
[6] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.
[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] C. Beeri and R. Ramakrishnan. 1987. On the Power of Magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '87)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/28659.28689
[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] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.
[13] 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
[14] Radu Ciucanu and Dan Olteanu. 2016. *Worst-Case Optimal Join at a Time*. Technical Report. Department of Computer Science, University of Oxford.
[15] Rina Dechter and Judea Pearl. 1987. Network-based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* 34, 1 (1987), 1–38. https://doi.org/10.1016/0004-3702(87)90002-6
[16] Rina Dechter and Judea Pearl. 1989. Tree Clustering for Constraint Networks. *Artif. Intell.* 38, 3 (April 1989), 353–366. https://doi.org/10.1016/0004-3702(89)90037-4
[17] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797
[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[19] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2nd ed.). Prentice Hall Press, USA.
[20] 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
[21] 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
[22] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2000. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence* 124, 2 (2000), 243–282. https://doi.org/10.1016/S0004-3702(00)00078-3
[23] 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
[24] 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
[25] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.
[26] Goetz Graefe and William J McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 209–218.
[27] Xiao Hu and Ke Yi. 2016. Towards a Worst-Case I/O-Optimal Algorithm for Acyclic Joins. 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, 135–150. https://doi.org/10.1145/2902251.2902292
[28] 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.
[29] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. 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, 91–106. https://doi.org/10.1145/2902251.2902293
[30] Stasys Jukna. 2011. *Extremal Combinatorics: With Applications in Computer Science* (2nd ed.). Springer Publishing Company, Incorporated.
[31] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible Caching in Trie Joins. *EDBT* (2017). https://openproceedings.org/2017/conf/edbt/paper-131.pdf
[32] 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
[33] Phokion G. Kolaitis and Moshe Y. Vardi. 1998. Conjunctive-Query Containment and Constraint Satisfaction. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 205–213. https://doi.org/10.1145/275487.275511
[34] David Maier. 1983. *The Theory of Relational Databases*. Computer Science Press. http://web.cecs.pdx.edu/%7Emaier/TheoryBook/TRD.html
[35] 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
[36] Yoon-Min Nam Nam, Donghyoung Han Han, and Min-Soo Kim Kim. 2020. SPRINTER: A Fast n-Ary Join Query Processing Method for Complex OLAP 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, 2055–2070. https://doi.org/10.1145/3318464.3380565
[37] Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. 2020. Optimal Joins Using Compact Data Structures. In *23rd International Conference on Database Theory (ICDT 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 155)*, Carsten Lutz and Jean Christoph Jung (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:21. https://doi.org/10.4230/LIPIcs.ICDT.2020.21
[38] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. 2014. Beyond Worst-Case Analysis for Joins with Minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/2594538.2594547
[39] 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
[40] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (Sept. 2016), 5–16. https://doi.org/10.1145/3003665.3003667
[41] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1, Article 2 (March 2015), 44 pages. https://doi.org/10.1145/2656335
[42] Anna Pagh and Rasmus Pagh. 2006. Scalable Computation of Acyclic Joins. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Chicago, IL, USA) *(PODS '06)*. Association for Computing Machinery, New York, NY, USA, 225–232. https://doi.org/10.1145/1142351.1142384
[43] S. Russell and P. Norvig. 2010. *Artificial Intelligence: A Modern Approach* (third ed.). Prentice Hall, Upper Saddle River, NJ. http://aima.cs.berkeley.edu/
[44] 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
[45] 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.
[46] Abraham Silberschatz, Henry F. Korth, and Shashank Sudarshan. 2019. *Database System Concepts* (7th ed.). McGraw-Hill New York.
[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] Susan Tu and Christopher Ré. 2015. DunceCap: Query Plans Using Generalized Hypertree Decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 2077–2078. https://doi.org/10.1145/2723372.2764946
[49] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland,*

OR, USA], June 14-19, 2020, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 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] Sungheun Wi, Wook-Shin Han, Chuho Chang, and Kihong Kim. 2020. Towards Multi-Way Join Aware Optimizer in SAP HANA. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3019–3031. https://doi.org/10.14778/3415478.3415531

[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

A PROOF OF THEOREM 3.1

Proof by induction on the height of G_Q. Base case. Suppose the height of G_Q is 0. Claim trivially holds. Suppose the claim holds for all queries whose height of $G_Q < h$. We want to show the claim holds for height of G_Q equals h. We want to show $J_1 = R_1 \bowtie \ldots \bowtie R_k$ and there is no dangling tuples in any intermediate result during computation. $(\ldots((R_1^* \bowtie R_2') \bowtie R_3') \ldots \bowtie R_m')$ equals to $R_1 \bowtie R_2' \bowtie \ldots \bowtie R_m'$.

$$
\begin{aligned}
J_1 &= (\ldots((R_1^* \bowtie R_2') \bowtie R_3') \ldots \bowtie R_m') \bowtie J_2 \bowtie \ldots \bowtie J_m \\
&= R_1 \bowtie R_2' \bowtie \ldots \bowtie R_m' \bowtie J_2 \bowtie \ldots \bowtie J_m \\
&= R_1 \bowtie (R_2' \bowtie J_2) \bowtie (R_3' \bowtie J_3) \bowtie \ldots \bowtie (R_m' \bowtie J_m) \\
&= R_1 \bowtie J_2 \bowtie J_3 \bowtie \ldots \bowtie J_m \\
&= R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k
\end{aligned}
$$

The last step because $J_2, \ldots, J_m$ are subtrees of G_Q and they are disjoint. To show there is no dangling tuple, pick R_1, R_j and R_i where R_j is a child of R_1 and R_i is a child of R_j. During $P_{Q,1}$, $R_1 \ltimes (R_j \ltimes R_i)$ is executed. Because G_Q is a join tree, R_1, R_j, R_i share common attributes. If there is a dangling tuple, it has to happen after $R_1 \bowtie R_j$. However this is not possible because $R_1 \bowtie R_j$ after $P_{Q,1}$ equals to $(R_1 \ltimes (R_j \ltimes R_i)) \bowtie (R_j \ltimes R_i)$, which is $(R_1 \bowtie R_j) \ltimes R_i$. By induction assumption, no dangling tuple when join relations in subtree rooted in R_i. Since R_j and R_i are picked arbitrarily, the theorem holds.

B PROOF OF LEMMA 5.5

Since the plan is in clean state, by Lemma 5.1, we have $R_{u-1}' \subseteq H_{u-1}'$. The query plan is created from a join tree, and by Theorem 3.1 there has to be some tuple in R_{u-1}' that can join with some tuple(s) in J_u^*. To show the resulting tuple is not a dangling tuple, we proceed with a proof by contradiction. Let $J_{u-1} = J_u^* \bowtie H_{u-1}'$ and $J = R_1 \bowtie \ldots \bowtie R_k$. Suppose a dangling tuple exists. That is, there exists $t_1 \in J_{u-1}$ such that there is no $t_2 \in J$ with $t_1[attr(J_{u-1}) \cap attr(J)] = t_2[attr(J_{u-1}) \cap attr(J)]$. Since $attr(J_{u-1}) \cap attr(J) = attr(J_{u-1})$, there is no $t_2 \in J$ with $t_1[attr(J_{u-1})] = t_2[attr(J_{u-1})]$. Then, it is sufficient to show there is no $t_2 \in J_u^* \bowtie R_{u-1}$ with the condition holding. Since $t_1 \in J_u^* \bowtie H_{u-1}'$, the assumption implies that there exists $t_1 \in J_u^* \bowtie H_{u-1}'$ such that $t_1 \notin J_u^* \bowtie R_{u-1}$. However, this is not true because $J_u^* \bowtie H_{u-1}' \subseteq J_u^* \bowtie R_{u-1}$.

C PROOF OF LEMMA 5.6

We will consider three possible cases.

Case 1. Suppose the query execution is already in the clean state at the beginning of the evaluation. Base case $u = k$. By Lemma 5.1, $R_k = R_k^*$ and the tuple returned from $\bowtie_k$ is in J_k^*. Assume the lemma holds for $u = i$. We show that lemma holds for $u = i - 1$. By induction, the assumption implies that $\bowtie_{i-1}$'s r_{outer} belongs to J_i^*. By Lemma 5.5, the joined tuple between r_{outer} and a tuple in H_{i-1}' cannot be dangling tuple. Thus, tuple produced by $\bowtie_{i-1}$ from Algorithm 5.3 Line 12 is in J_{i-1}^*. In addition, with Lemma 5.3, the tuple returned from Algorithm 5.3 Line 6 is in J_{i-1}^*. The lemma holds.

Case 2. Suppose the clean state happens at $u = k$. Consider the base case $u = k$. The assumption indicates that the clean state is formed right after Algorithm 5.5 Line 2 is executed. By Lemma 5.1, $R_k = R_k^*$ and the tuple returned from $\bowtie_k$ is in J_k^*. Assume the lemma holds for $u = i$. We show the lemma holds form $u = i - 1$. Since the clean state happens at $u = k$, Algorithm 5.5 Line 3 will eventually cause $\bowtie_{i-1}$'s r_{outer} reassigned. By induction assumption, $\bowtie_{i-1}$'s r_{outer} will be from J_i^*. By Lemma 5.3, l will be initialized and by Lemma 5.5, we know the joined tuple returned from $\bowtie_{i-1}$ is in J_{i-1}^*.

Case 3. Suppose the clean state happens at $u = i$ where $i \in [k-1]$. This happens after Algorithm 5.4 Line 3 is executed. Base case $u = k$. The assumption indicates that the tuple returned by $\bowtie_k$ is already in J_k^* because otherwise, the clean state will happen at $u = k$. Assume the lemma holds for $u = j$. We show the lemma holds for $u = j - 1$. Using a similar argument as Case 2, the lemma holds.