TreeTracker Join: Turning the Tide When a Tuple Fails to Join

Anonymous Author(s)

ABSTRACT

Many important query processing methods proactively use semijoins or semijoin-like filters to delete dangling tuples, i.e., tuples that do not appear in the final query result. Semijoin methods can achieve formal optimality but have high upfront cost in practice. Filter methods reduce the cost but lose the optimality guarantee.

We propose a new join algorithm, TreeTracker Join (TTJ), that achieves the data complexity optimality for acyclic conjunctive queries (ACQs) without semijoins or semijoin-like filters. TTJ leverages *join failure* events, where a tuple from one of the relations of a binary join operator fails to match any tuples from the other relation. TTJ starts join evaluation immediately and when join fails, TTJ identifies the tuple as dangling and prevents it from further consideration in the execution of the query. The design of TTJ exploits the connection between query evaluation and constraint satisfaction problem (CSP) by treating a join tree of an ACQ as a constraint network and the query evaluation as a CSP search problem. TTJ is a direct extension of a CSP algorithm, TreeTracker, that embodies two search techniques *backjumping* and *no-good*. We establish that join tree and plan can be constructed from each other in order to incorporate the search techniques into physical operators in the iterator form. We compare TTJ with hash-join, a classic semijoin method: Yannakakis's algorithm, and two contemporary filter methods: Predicate Transfer and Lookahead Information Passing. Favorable empirical results are developed using standard query benchmarks: JOB, TPC-H, and SSB.

CCS CONCEPTS

• **Information systems → Join algorithms**.

KEYWORDS

optimal join algorithm, join operator, acyclic conjunctive queries, join ordering, sideway information passing

ACM Reference Format:

Anonymous Author(s). 2025. TreeTracker Join: Turning the Tide When a Tuple Fails to Join. In *In Proceedings of the 2025 International Conference on Management of Data (SIGMOD '25)*. ACM, New York, NY, USA, 14 pages. https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

1 INTRODUCTION

Removing *dangling tuples*, tuples that do not contribute to the final output of a query [26], has been central in improving both formal and practical join query execution speed [9, 13, 18, 22, 23, 25, 30, 33, 36, 43, 45, 49, 53, 55, 57, 67–69, 71]. However, a trade-off exists as the cost of dangling tuples removal may offset the join performance improvement. Yannakakis's algorithm (YA) is a representative of semijoin methods [13, 18, 57, 67–69] for acyclic conjunctive queries (ACQs) evaluation. YA executes a sequence of semijoins called full reducer (F_Q) as a preprocessing step and removes the dangling tuples from the input relations completely before join evaluation [13, 67]. As a result, YA provides optimal data complexity guarantee. However, in practice, using semijoins introduces high upfront costs [29, 57, 62]. On the other hand, filter methods [22, 23, 25, 30, 32, 33, 36, 45, 49, 53, 55, 71] usually trade off optimal data complexity gurantee for reduction of dangling tuple removal cost by replacing semijoins with semijoin-like filter structures, e.g., Bloom filters [15] and removing dangling tuples by proactively checking base relations against filters. Efficient filter implementation allows these methods to work well in practice. Both semijoin and filter methods are *eager* approaches because they preemptively remove dangling tuples, aiming to prevent possible join failures (events where a tuple from one of the relations of a binary join operator fails to match any tuples from the other relation) from happening. Those methods rely on the efficient amortization of the upfront cost, incurred by dangling tuple removal, over the resulting join time reduction. If few dangling tuples exist, the upfront cost of the methods cannot be sufficiently amortized and the cost of dangling tuple reduction is more likely to outweigh its benefits. In an extreme case where no dangling tuples exist in the input relations, dangling tuple removal operations induce extra costs with no benefits. Common existing mitigations of this problem rely on heuristics such as disabling the filters based on selectivity estimation of the underlying relations [22, 25, 55], which require workload-specific assessment on the trade-off between the execution cost and the potential speed improvement.

TreeTracker Join (TTJ) is the first join algorithm that leverages join failure events to remove dangling tuples with minimal overhead while maintaining the optimal data complexity for ACQs. TTJ is a *lazy* approach. The signature feature of TTJ is to start join evaluation immediately without any preprocessing and perform two additional operations only on join failure: (1) identifying which tuple from which relation (*guilty relation*) causes a join failure at another relation (*detection relation*), and (2) subsequently removing the tuple from the guilty relation. The goal of TTJ is to remove a sufficient number of dangling tuples in the minimal amount of time to achieve a satisfactory level of join time reduction. Comparing with YA, TTJ does not aim to remove all dangling tuples, but the optimal guarantee still holds.

Fundamentally, TTJ exploits the equivalence between *constraint satisfaction problem* (CSP) and conjunctive query processing [16, 39] by treating query evaluation as a search problem. The intuition is that *join tree* $\mathcal{T}_Q$, the graph representation of ACQ, can be interpreted from CSP perspective as a *constraint network*. For example, consider a binary join between $A(x, y)$ and $B(y, z)$, which is acyclic and its $\mathcal{T}_Q$ is $A - B$. Interpreting $\mathcal{T}_Q$ as a constraint network, we

SIGMOD '25, xxxx xx–xx, 2025, xxxx, xx

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

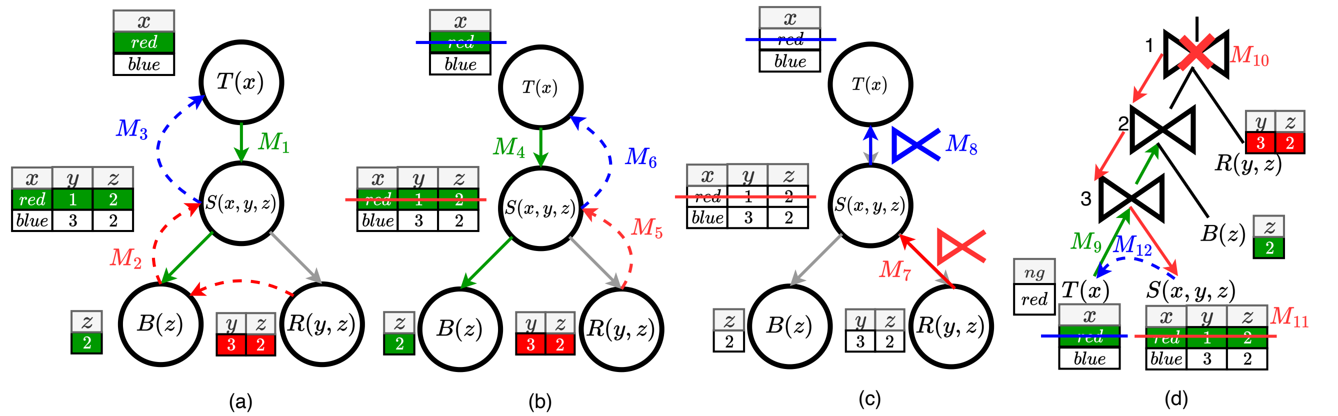

Figure 1: Illustration of the identification and removal of two dangling tuples by different algorithms: (a) join evaluation viewed as solving a CSP; (b) TTJ using CSP search techniques (backjumping and no-good) on the join tree $\mathcal{T}_Q$; (c) Yannakakis's algorithm (YA); and (d) TTJ packed into physical operators on a left-deep query plan. *We explain the details in Example 1. M_i are execution moments referenced throughout the paper.*

view both A and B as variables. Tuples in each relation are possible *assignments* to each variable. Our goal is to find all possible assignments to A and B such that *constraint* $A.y = B.y$ is satisfied. $\mathcal{T}_Q$, when viewed as a constraint network, can be evaluated using search techniques such as *backjumping* and *no-good*, which are commonly-used in both database [6, 19, 34] and AI communities [10, 21, 27]. TTJ is a direct extension of a CSP algorithm, TreeTracker [10], that embodies the aforementioned two search techniques. We show $\mathcal{T}_Q$ and query plan can be easily constructed from each other. Thus, the aforementioned search techniques can be integrated into a query plan. In this paper, we directly encode the two search techniques into physical operators in iterator interface [26], utilizing the form of *sideways information passing (SIP)*. To help understand how TTJ works, we illustrate the CSP view of query evaluation, and the unique features of TTJ using Example 1.

Example 1. Consider a join of 4 relations $T(x)$, $S(x, y, z)$, $B(z)$, and $R(y, z)$ with the database instance shown in Figure 1. All four plots show how the same two dangling tuples from the database instance are identified and removed by different algorithms.

(a) presents how evaluating a $\mathcal{T}_Q$ can be viewed as solving a CSP by recursively assigning variables one by one until all variables are successfully assigned. The evaluation starts to assign T with a tuple from its instance $T(red)$ and then moves on to S (moment M_1). Since $S(red, 1, 2)$ agrees with $T(red)$ on attribute x, $S(red, 1, 2)$ can be assigned to S. This assignment is the same as obtaining a join result $(red, 1, 2)$ for $T \bowtie S$. The process continues to B and assigns B with $B(2)$. $R(3, 2)$ cannot be assigned to R given all the previous assignments because $(y, z) = (3, 2)$ in $R(3, 2)$ but $(y, z) = (1, 2)$ in $S(red, 1, 2)$. Since no other tuples from R can be assigned, the search process has to *backtrack* to B to try a different value given the existing assignments on T and S. Since no other tuples from B can be assigned, the search backtracks to S at M_2. The same behavior repeats at S and the process further backtracks to T at M_3. Then, T is assigned with the next tuple $T(blue)$ and the process continues. When all variables are successfully assigned, we obtain one solution to the CSP by joining all the current assignments to the variables. The solution to the CSP is exactly a join result to the query. The search process for the next solution continues until all the solutions to the CSP are found.

(b) shows how TTJ improves the solving process in (a) with the two search techniques and removes two dangling tuples. The process (M_4) is identical to (a) until it fails to assign a tuple to R. Unlike (a) where the process backtracks to the previously assigned variable B, TTJ directly *backjumps* to S (M_5), the parent of R in $\mathcal{T}_Q$. Relations skipped due to backjumping are called *backjumped relations*, e.g., B. Once the search backjumps to S, the current assignment to S is marked as *no-good*, i.e., $S(red, 1, 2)$ is a dangling tuple. TTJ removes $S(red, 1, 2)$ from the instance of S and the removed tuple will not be considered again for future assignments. Since no other tuples from S can be assigned, backjump happens again (M_6) and $T(red)$ is removed.

(c) highlights how YA removes the same dangling tuples as TTJ in a different way. YA executes the full reducer F_Q, a sequence of semijoins, before join starts: At M_7, $S' = S \ltimes R$ and $S(red, 1, 2)$ is removed. Then, at M_8, $T \ltimes S'$ and $T(red)$ is removed. Unlike TTJ that removes dangling tuples while performing join, YA removes all dangling tuples before join starts.

(d) illustrates the same join process as (b) on a left-deep query plan using demand-driven pipelining with operators implemented in iterator interface consisting of `open()` and `getNext()`. The evaluation starts with recursive `open()` calls on the join operators and builds hash tables on S, B, and R. To obtain the first query result, the join process first calls $\bowtie_1$'s `getNext()`, which calls its left child $\bowtie_2$'s `getNext()`, and such pattern repeats until the left most relation T's `getNext()` is called and returns $T(red)$ (M_9). $\bowtie_3$ probes into $\mathcal{H}_S$, the hash table on S, and finds a matching tuple $S(red, 1, 2)$. The joined result $(red, 1, 2)$ is returned to $\bowtie_2$. Then, the matching tuple $B(2)$ from $\mathcal{H}_B$ joins with $(red, 1, 2)$ and the joined result $(red, 1, 2)$ is returned to $\bowtie_1$. Probing into hash tables to find a matching tuple is the same as assigning a tuple to a variable in CSP. No tuples from $\mathcal{H}_R$ join with $(red, 1, 2)$ (M_{10}); hence, join fails at R and R is the detection relation. Thus, TTJ performs backjumping making additional method calls to reset the evaluation flow to

S, the guilty relation, because S is the parent of R in $\mathcal{T}_Q$. Subsequently, $S(red, 1, 2)$ is removed from $\mathcal{H}_S$ (M_{11}), which is logically equivalent to removing the tuple from the instance of S. Since no tuples from S join with $T(red)$, TTJ backjumps to T and implicitly removes $T(red)$ by adding it to a no-good list ng (M_{12}). The no-good list will be used in future steps to filter out dangling tuples from T.

The rest of the paper fills the missing details from Example 1 such as how to construct $\mathcal{T}_Q$ from a query plan (and vice versa), how TTJ packs backjumping and no-good techniques into a physical operator in the form of SIP, and formally show the correctness and optimality guarantee of TTJ. In summary, this paper makes the following contributions:

(1) We use CSP search techniques to design a lazy join algorithm TTJ that removes dangling tuples if they cause join failures (§ 3).
(2) We propose an algorithm to construct join tree from query plan, and vice versa (§ 3.1).
(3) We formally show TTJ works correctly and runs optimally in data complexity for ACQ (§ 4).
(4) We deduce a general condition called clean state that enables optimal evaluation of ACQ while permitting the existence of dangling tuples (§ 4).
(5) We conduct extensive experiments to compare TTJ with four baseline algorithms on three benchmarks and perform detailed analysis to understand the features of TTJ (§ 5).

2 PRELIMINARIES

We review related background on acyclic conjunctive query evaluation, formulate the problem, and summarize the notation used in this paper .

2.1 Acyclic Conjunctive Query Evaluation

We consider a relational database consisting of k relations under bag semantics. A *full conjunctive query* (CQ) is a natural join of k relations:

$$Q(\boldsymbol{a}) = R_1(\boldsymbol{a_1}) \bowtie R_2(\boldsymbol{a_2}) \bowtie \ldots \bowtie R_k(\boldsymbol{a_k}) \quad (1)$$

For each relation $R_i(\boldsymbol{a_i})$, $\boldsymbol{a_i}$ is a tuple of variables called *attributes*. We define $attr(R_i) = \boldsymbol{a_i}$. Q is full because $\boldsymbol{a}$ includes all the attributes appearing in the relations, i.e., $attr(Q) = \bigcup_{u=1}^{k} attr(R_u)$.

Query graph. The literature contains a number of different graph representations of Q. The most common choice is hypergraph [28, 46]. To better emphasize the connection between CSP and query evaluation, we use an equivalent [21] alternative, *query graph* [17] (also known as *join graph* [66][1], *dual constraint graph* [21], or *complete intersection graph* [42]). The *query graph* of Q is a graph where there is a bijection between nodes in the graph and relations in the query. Two nodes v_1, v_2 are adjacent if their corresponding relations R_1, R_2 satisfy $attr(R_1) \cap attr(R_2) \neq \emptyset$. For clarity, we use the relations to label the nodes in the query graph.

Join Tree. Q is *acyclic* if its query graph contains a spanning tree called *join tree* $\mathcal{T}_Q$, which satisfies the *connectedness property* [11, 21]: for each pair of distinct nodes R_i, R_j in the tree and for every common attribute a between R_i and R_j, every relation on the path between R_i and R_j contains a. For the rest of the paper, we assume Q is a full acyclic CQ (ACQ). For ACQ, one can find a maximum-weight spanning tree from the query graph, where the weight of an edge (R_i, R_j) is $|attr(R_i) \cap attr(R_j)|$. Such tree is guaranteed to be a join tree [42]. A *rooted join tree* is a join tree converted into a directed tree with one of the nodes chosen to be the root. We assume $\mathcal{T}_Q$ is a rooted join tree.

Query Plan. Physical evaluation of ACQ is commonly done using query plan. A *query plan* is a binary tree, where each internal node is a join operator $\bowtie$, and each leaf node is a scan operator (we use table scan by default) associated with one of the relations $R_i(\boldsymbol{a_i})$ in Query (1). The plan is a *left-deep query plan*, or *left-deep plan*, if the right child of every join operator is a leaf node [51]. For example, $((T \bowtie S) \bowtie B) \bowtie R$ in Figure 2 (c) is a left-deep plan. In the paper, we focus on the left-deep plan and expand to the other plan shape in [2]. As a shorthand [64], we represent a left-deep plan, labeled from bottom to top, $(\ldots((R_k \bowtie R_{k-1}) \bowtie R_{k-2}) \ldots) \bowtie R_1$ as $[R_k, R_{k-1}, \ldots, R_1]$.

Example 2. Consider an ACQ

$$Q(x, w, z) = T(x) \bowtie S(x, y, z) \bowtie B(z) \bowtie R(y, z) \quad (2)$$

Figure 2 illustrates query graph, join tree, and query plan of Q. $\mathcal{T}_Q$ in (b) is obtained from the query graph in (a) by removing edge (B, R). B and R satisfy the connectedness property because S, the only relation on the path between B and R, also shares their common attribute z. From CSP perspective, removing edge (B, R) from the query graph does not impact the query result because the constraint $B.z = R.z$ is enforced via an alternate path $B - S - R$, i.e., $B.z = S.z \wedge S.z = R.z$.

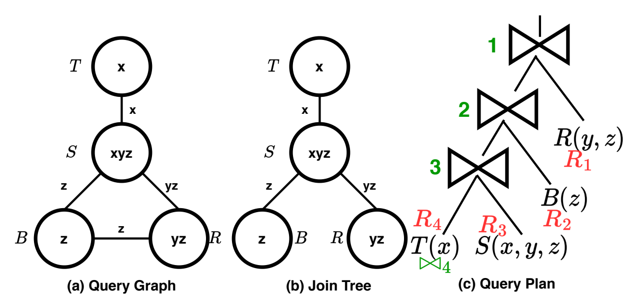

Figure 2: (a) query graph, (b) join tree , and (c) query plan of Q in Example 2. $R_1, \ldots, R_4$ show the relation numbering and $\bowtie_1, \bowtie_2, \bowtie_3, \bowtie_4$ denote the join operator numbering. $\bowtie_4$ represents the table scan operator associated with the leftmost relation R_4, which is T in this example.

Complexity measurement. We assume a standard RAM complexity model [5]. Following the convention of research in the formal study of conjunctive query processing [4, 38, 59], we use data complexity (big-$\mathcal{O}$ notation) as the measure of TTJ theoretical performance, which assumes that the size of a query, k, is a constant, but data size n varies [8]. We also determine TTJ performance in combined complexity [63] (big-O notation), which considers both k and n as variables. Under data complexity, the lower bound of any join algorithm is $\Omega(n+r)$ [59] (r is the output size) 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] Join graph is defined in CSP and database theory with a slightly different definition: a spanning subgraph of query graph that satisfies the connectedness property [21, 42].

Physical Operators. Operators in the query plan of Q are physical operators, commonly implemented in an iterator interface [26] consisting of `open()`, `getNext()`, and `close()`. `open()` prepares 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. In this paper, evaluation of a query plan is done using *demand-driven pipelining* (or *pipelining*): it first calls `open()` of each operator and then keeps calling `getNext()` of the root join operator of the plan, which further recursively calls `getNext()` of the rest of the operators, until no more tuples are returned [56].

2.2 Problem Definition

With the above background, we are ready to define the problem that TTJ tries to solve.

Problem. Given an ACQ Q, we want to evaluate a left-deep query plan of Q consisting of physical join operators implemented in iterator interface using demand-drive pipelining with formal optimality guarantee and practical efficiency.

2.3 Baselines

We compare TTJ with in-memory hash-join (HJ), one classic semijoin method: Yannakakis's algorithm (YA), and two representative filter methods: Lookahead Information Passing (LIP) and Predicate Transfer (PT). We introduce each of them in order.

HJ evaluates Q using pipelining on a left-deep plan with in-memory hash-join operators [30]. In `open()`, each hash-join operator builds a hash table $\mathcal{H}$ from its right child R_{inner}. In `getNext()`, a tuple t from the left child of the join operator, R_{outer}, probes into $\mathcal{H}$ to find a set of joinable tuples denoted as *MatchingTuples*. `getNext()` returns the join between t and the first tuple from *MatchingTuples*. The join between t and the rest of the tuples will be returned in the subsequent `getNext()` calls.

YA [67] is an optimal join algorithm for ACQ. The algorithm consists of two phases: a *full reducer phase* and a *join phase*. In the full reducer phase, YA makes two passes over $\mathcal{T}_Q$. The first pass, called *reducing semijoin program* [13] HF_Q, traverses the join tree bottom-up and applies $R_p \ltimes R_c$ where R_p is a parent relation and R_c is one of its children. The possibly reduced R_p further semijoins with its other children. The resulting relations after HF_Q are denoted as R'_i. For example, in Figure 1 (c), two semijoins $S' = S \ltimes R$ and $T' = T \ltimes S'$ are part of the bottom-up pass. In the second pass, the algorithm traverses $\mathcal{T}_Q$ top-down applying $R'_c \ltimes R'_p$ [2]. The fully reduced relations are denoted as R^*_i for $i \in [k]$ [3] and they are free of dangling tuples. In the join phase, YA makes the third pass of $\mathcal{T}_Q$ to produces the join output by again traversing $\mathcal{T}_Q$ bottom-up and performing pairwise joins.

LIP [25, 70, 71] leverages a set of Bloom filters to evaluate star schema queries consisting of a fact table and dimension tables. In `open()`, LIP computes filters from R_{inner} of each join operator and passes those filters downwards along the left-deep plan to the fact table, which is the left-most relation of the plan. In `getNext()` of the left-most table scan operator, LIP checks the tuples from the fact table against the filters and propagates those pass the check upwards along the plan.

PT [66] is the state-of-the-art filter method that generalizes the idea of LIP to queries not limited to star schema queries. Similar to YA, PT divides query evaluation into two phases. First, in predicate transfer phase, PT passes filters over the predicate transfer graph, a directed acyclic graph built from the query graph, of a query in two directions: forward and backward, which is similar to the first two passes over $\mathcal{T}_Q$ in YA. Relations are gradually reduced as filters are being passed. Once the predicate transfer phase is done, the join phase begins where the reduced relations are joined.

[2] $R_c \ltimes R'_p$ if R_c is a leaf node because leaf nodes are not reduced in the first pass.
[3] $[k]$ is a shorthand for $1, \dots, k$

2.4 Notation

Table 1: Summary of common notation

Notation	Definition
Q	a full acyclic CQ
k	number of relations in Q
n	maximum size of the input relations in Q
r	query output size
$\mathcal{T}_Q$	rooted join tree. See Figure 2 (b).
$\mathcal{P}_Q$	a left-deep query plan using TTJ (§ 3)
R_i for $i \in [k]$	relations in $\mathcal{P}_Q$. Left-most relation is R_k. See Figure 2 (c).
$\bowtie_i$ for $i \in [k]$	join operators in $\mathcal{P}_Q$. $\bowtie_1$ is the root operator. $\bowtie_k$ is the table scan operator of R_k. See Figure 2 (c).
$[R_k, R_{k-1}, \dots, R_1]$	a query plan $(\dots((R_k \bowtie R_{k-1}) \bowtie R_{k-2})\dots) \bowtie R_1$
J^*_u	join of relations $R_k, R_{k-1}, \dots, R_u$
$t[a]$	$t[a] = \pi_a(t)$ for tuple t, attribute a, and projection π
$ja(R, S)$	$attr(R) \cap attr(S)$
$R(3, 2)$	tuple $(3, 2) \in R$
$jav(t, R, S)$	join-attribute value $t[attr(R) \cap attr(S)]$
R_{inner}	right child of $\bowtie_i$
R_{outer}	left child of $\bowtie_i$
$\mathcal{H}_R$ (or $\mathcal{H}_i$)	hash table built from R (or associated with $\bowtie_i$)
MatchingTuples	the list of tuples with the same *jav* in a hash table
ng	no-good list, a filter in TTJ scan
$\mathbb{R}$	physical aspects of R, i.e., a bag of tuples R contains

We summarize the notation used in the paper in Table 1. We omit standard relational algebra notation in the table, e.g., antijoin $\overline{\ltimes}$ and semijoin $\ltimes$. We further define some terminologies used throughout the paper. We call a relation *internal* if it appears as an internal node [20, 52] in $\mathcal{T}_Q$. For relations corresponding to non-root internal nodes of $\mathcal{T}_Q$, we call them *internal*$^\circ$ *relations*. Similarly, a *leaf relation* means the relation appears as a leaf node in $\mathcal{T}_Q$. The *root relation* is defined accordingly. Depending on context, we adapt the following language: 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 $\bowtie_i$ (*dead-end* in CSP [21]), we call it a *join fails at* $\bowtie_i$, a *join failure happens at* $\bowtie_i$, or *join fails at* R_i. In such case, R_i is called the

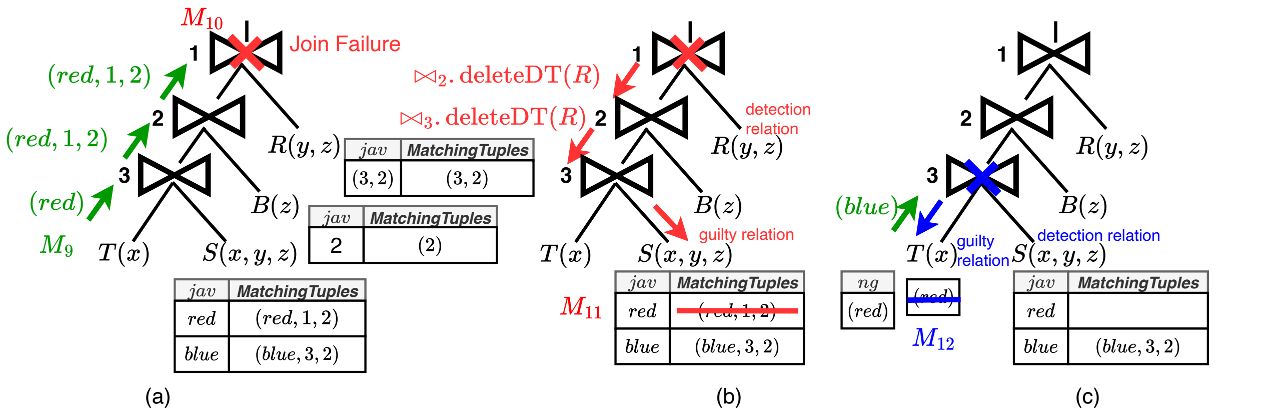

Figure 3: (a) Join fails at $\bowtie_1$. (b) A series of `deleteDT(R)` is called, which leads to the removal of $S(red, 1, 2)$ from hash table $\mathcal{H}_S$. (c) Join further fails at $\bowtie_3$, which puts $T(red)$ to *ng*.

detection relation (*dead-end variable* in CSP [21]). $\bowtie_i$ is called the *detection operator*. We call the join operator the *removal operator* if its R_{inner} is the parent of the detection relation for a join failure in $\mathcal{T}_Q$. Such R_{inner} is the *guilty relation* (*culprit variable* in CSP [21]). For example, for the join failure happens at $\bowtie_1$ in Figure 1 (d), the detection relation is R and the detection operator is $\bowtie_1$. S is the guilty relation and $\bowtie_3$ is the removal operator.

3 TREETRACKER JOIN OPERATORS

Algorithms 3.1 and 3.2 show the formal definition of TTJ. Algorithm 3.1 defines each join operator in a left-deep plan. Algorithm 3.2 defines TTJ *scan*, which replaces the normal left-most table scan operator; the rest of the table scan operators in the plan remains unchanged. We use $\mathcal{P}_Q$ to denote the left-deep plan using TTJ. We are now ready to work out Example 1 in full details to highlight the salient features of TTJ mentioned in § 1. We expand Figure 1 (d) into Figure 3. All line numbers reference Algorithm 3.1 by default unless noted otherwise.

The following three examples show the execution moments in the first `getNext()` call after `open()` of the pipelining evaluation that leads to the removal of two dangling tuples. Example 3 shows that TTJ does not schedule any semijoins or semijoin-like filters before query evaluation. The evaluation flow is identical to HJ when no join failure happens.

Example 3 (M_9 in Figures 1 and 3). After plan evaluation begins, the recursive `getNext()` calls start with $\bowtie_1$ and end with T's TTJ scan operator (Line 4 Algorithm 3.2), which returns $T(red)$. The *jav* $(x : red)$ is used to look up $\mathcal{H}_S$ (Line 15). Since $T(red)$ joins with $S(red, 1, 2)$, the resulting tuple $(red, 1, 2)$ is further propagated to $\bowtie_2$, which probes into $\mathcal{H}_B$ and finds $B(2)$ joinable. The join result $(red, 1, 2)$ is further passed to $\bowtie_1$.

Example 4 shows how the backjumping idea from CSP (specifically, *graph-based backjumping* [21]) shown in Example 1 is integrated into physical operators in $\mathcal{P}_Q$. To do so, we enhance the iterator interface with one more method `deleteDT()` and implements backjumping as a series of `deleteDT()` calls [4] from the detection operator to the removal operator corresponding to a join failure. `deleteDT()`, under the form of SIP, sends the reference of the detection relation from the detection operator to the removal operator in a fashion that is not explicitly indicated by the plan.

Example 4 (M_{10} and M_{11} in Figures 1 and 3). Since $(red, 1, 2)$ cannot join with any tuples from $\mathcal{H}_R$, the goal of TTJ is to backjump to the guilty relation S and remove the last returned tuple, $S(red, 1, 2)$, from $\mathcal{H}_S$. To do so, $\bowtie_2$`.deleteDT(R)` is called from Line 20 first. Since $\bowtie_2$'s R_{inner}, B, is not the parent of R in $\mathcal{T}_Q$ (Line 23), Line 27 is called, e.g., $\bowtie_3$`.deleteDT(R)`. In $\bowtie_3$'s `deleteDT()`, since S is the parent of R (Line 23), Line 24 is executed: $S(red, 1, 2)$ is removed from $\mathcal{H}_S$.

Example 4 shows that removing tuples from internal° relations [5] is implemented as removing the tuples from their index representations. Example 5 illustrates another CSP technique, *no-good list* (*ng*), that TTJ incorporates to filter out dangling tuples from the left-most relation R_k.

Example 5 (M_{12} in Figures 1 and 3). Removal of $S(red, 1, 2)$ causes $T(red)$ to become dangling. TTJ adds it to *ng*, effectively removing it from T. After removing $S(red, 1, 2)$, `getNext()` of $\bowtie_3$ is called (Line 29). Since *MatchingTuples* is now empty and $r_{outer} = T(red)$, Line 15 is executed. No tuples from S joins with $T(red)$. Thus, T`.deleteDT(S)` is called (Line 20) and Algorithm 3.2 Line 10 adds *jav* $(x : red)$ to *ng*. Once *ng* is non-empty, it will work like a filter to prevent future dangling tuples with the same *jav* from returning to $\bowtie_3$. `getNext()` of T is called (Algorithm 3.2 Line 11). The next tuple $T(blue)$ then probes into *ng* (Algorithm 3.2 Line 6). Since T has only one child S, *jav* $(x : blue)$ is computed and it is not in *ng*. Thus $T(blue)$ is safe to further propagate upwards towards $\bowtie_3$.

3.1 Construction of Query Plan or Join Tree

TTJ operates on a left-deep query plan, which represents the join order of the input relations of the query. In addition, TTJ requires a $\mathcal{T}_Q$ to find the parent of the detection relation, i.e., the guilty relation, for a join failure. Thus, if either the plan or the $\mathcal{T}_Q$ is missing, we need to construct it from the other one. A constraint exists for such construction to ensure TTJ can function correctly.

[4] We omit argument to `deleteDT()` when reference it generically.

[5] No tuples are removed from the leaf relations because they cannot be guilty relations, i.e., by leaf definition, they are not parent of any relations in $\mathcal{T}_Q$.

Algorithm 3.1: TTJ Join Operator

```
Purpose: An iterator returns, one at a time, the join result
         of R_outer and R_inner.
Output: A tuple t ∈ R_outer ⋈ R_inner
TTJOperator
  void open()
    // r_outer references a tuple from R_outer
    // MatchingTuples references a set of tuples from
       R_inner that are joinable with r_outer
    Initialize r_outer, MatchingTuples to nil
    R_inner.open()
    Build hash table H: Insert each tuple, r_inner, from
      R_inner into H using the join attribute value(s),
      jav(r_inner, R_outer, R_inner) as the key
    R_outer.open()
  Tuple getNext()
    if MatchingTuples ≠ nil ∧ MatchingTuples ≠ ∅ then
      // If there are more matching tuples left, return
         the join of r_outer and the next matching tuple
      if (aMatchingTuple ← MatchingTuples.next()
       ) ≠ nil then
        return the join of r_outer and
          aMatchingTuple
      // No matching tuples are left. Get a new r_outer
      r_outer ← R_outer.getNext()
      if r_outer = nil then return nil
    if r_outer = nil then r_outer ← R_outer.getNext()
    while r_outer ≠ nil do
      // Find tuples from R_inner joinable with r_outer
      MatchingTuples ←
       H.get(jav(r_outer, R_outer, R_inner))
      if MatchingTuples ≠ nil then
        aMatchingTuple ← MatchingTuples.next()
        return the join of r_outer and
          aMatchingTuple
      else
        // Join failure identified; start the
           backjumping to the guilty relation, parent
           of R_inner in T_Q
        r_outer ← R_outer.deleteDT(R_inner)
    return nil
  Tuple deleteDT(Detection Relation R)
    if R_inner is the parent of R in T_Q then
      // R_inner is the guilty relation; join failure was
         identified at R because the join between r_outer
         and aMatchingTuple was eventually returned to
         R and cannot join with any tuples from R
      Remove aMatchingTuple from MatchingTuples
       and H
    else
      // Has not reached the guilty relation for R;
         backjumping continues
      MatchingTuples ← nil
      r_outer ← R_outer.deleteDT(R)
      if r_outer = nil then return nil
    return getNext()
```

Algorithm 3.2: TTJ Table Scan Operator for R_k

```
Purpose: Table scan operator for R_k that returns tuples
         not in ng.
TTJScan
  void open()
    Initialize ng to an empty set
  Tuple getNext()
    while (t ← R_k.next()) ≠ nil do
      if jav(t, R_k, R_i) ∉ ng for all children R_i of R_k in
       T_Q then
        return t
    return nil
  Tuple deleteDT(Detection Relation R)
    // R_k is the guilty relation; t contributes to the
       tuple that caused the join failure at R
    Insert jav(t, R_k, R) into ng
    return getNext()
```

Since deleteDT() always sends a reference of the detection relation downwards along the plan, when the plan is missing, we need to construct a plan such that the guilty relation must sit below the detection relation. For the same reason, when $\mathcal{T}_Q$ is missing, we need to construct a $\mathcal{T}_Q$ such that for any detection relation in a plan, exactly one of the relations below it must be its parent in the tree. In this section we formalize the constraint and describe how to properly construct a $\mathcal{T}_Q$ or a plan given the other input.

Given a left-deep query plan, Definition 1 defines the aforementioned constraint on the $\mathcal{T}_Q$.

*Definition 1 (**join tree assumption**).* Suppose $\mathcal{P}_Q = [R_k, R_{k-1}, \ldots, R_1]$. TTJ assumes $\mathcal{T}_Q$ satisfies the following property: for a given relation R_i in $\mathcal{P}_Q$, its parent in $\mathcal{T}_Q$ is one of the relations $R_k, R_{k-1}, \ldots, R_{i+1}$. The root of $\mathcal{T}_Q$ is the left-most relation R_k.

Example 6. Consider $\mathcal{P}_Q$ in Figure 2 (c), B is labeled as R_2. TTJ expects that B's parent in $\mathcal{T}_Q$ has to be either R_3 or R_4. As shown in Figure 2 (b), B's parent is S, which corresponds to R_3. Thus, $\mathcal{T}_Q$ in (b) satisfies the assumption.

The next lemma states that we can easily construct a required $\mathcal{T}_Q$ from any left-deep query plan that does not have cross-product.

Lemma 3.1. *For any left-deep plan without cross-product for acyclic queries, there exists a $\mathcal{T}_Q$ satisfies the join tree assumption (Definition 1).*

We defer the construction step and proof to [2]. The key idea is as follows: We construct $\mathcal{T}_Q$ following the order of relations in $\mathcal{P}_Q$ from left to right. Suppose $R_k, \ldots, R_{j+1}$ are already added to $\mathcal{T}_Q$. For R_j, we want to find a relation R_i that is already in $\mathcal{T}_Q$ such that $attr(R_j) \cap (\bigcup_{u=j+1}^{k} attr(R_u)) \subseteq attr(R_i)$. Left-deep query plan without cross-product for acyclic queries guarantees such R_i exists. We add R_j in $\mathcal{T}_Q$ through an edge (R_i, R_j).

Example 7. Suppose $\mathcal{P}_Q = [R_3(x,y), R_2(x,y,z), R_1(y,z)]$. The left-most relation $R_3(x,y)$ has to be the root of $\mathcal{T}_Q$. For the next relation $R_2(x,y,z)$, since only R_3 is in $\mathcal{T}_Q$ and $attr(R_2) \cap attr(R_3) \subseteq$

$attr(R_3)$, we add edge (R_3, R_2). Now, both R_3 and R_2 are in $\mathcal{T}_Q$ and union of their attributes is $\{x, y, z\}$. Since $attr(R_1) \cap \{x, y, z\} \subseteq attr(R_2)$, we add edge (R_2, R_1). The final $\mathcal{T}_Q$ is $R_3 \to R_2 \to R_1$.

Example 8. Consider a cyclic query, $\mathcal{P}_Q = [R_3(a, b), R_2(b, c), R_1(c, a)]$, the classic triangle query. Let us try to construct $\mathcal{T}_Q$. $R_3(a, b)$ is the root. $R_2(b, c)$ connects R_3. $attr(R_3) \cup attr(R_2) = \{a, b, c\}$. But, $attr(R_1) \cap \{a, b, c\} \nsubseteq attr(R_2)$ and $attr(R_1) \cap \{a, b, c\} \nsubseteq attr(R_3)$. R_1 cannot be placed in $\mathcal{T}_Q$ to satisfy the connectedness property while keeping $\mathcal{T}_Q$ being a tree.

Example 9. $\mathcal{P}_Q = [T(x), R(y, z), B(z), S(x, y, z)]$ contains a cross-product due to $T(x), R(y, z)$. We cannot construct $\mathcal{T}_Q$ because $\mathcal{T}_Q$ is a subgraph of the query graph and the query graph does not contain (T, R) edge.

Definition 1 can be interpreted as a join order assumption, which defines the constraint on the plan.

Corollary 3.2 (join order view of Definition 1). *Given a $\mathcal{T}_Q$, TTJ assumes the order of relations in a left-deep query plan satisfies the following property: for a node R_i and its child R_j in $\mathcal{T}_Q$, R_i is before R_j in $\mathcal{P}_Q$, i.e., $\mathcal{P}_Q = [\dots, R_i, \dots, R_j, \dots]$.*

Construction of $\mathcal{P}_Q$ is straightforward: performing a top-down pass (not necessarily from left to right) of $\mathcal{T}_Q$.

Example 10. For $\mathcal{T}_Q$ in Figure 2 (b) with T as the root, both $\mathcal{P}^1_Q = [T, S, B, R]$ and $\mathcal{P}^2_Q = [T, S, R, B]$ are valid plans for TTJ.

3.2 Additional Practical Considerations

To use TTJ in production environment, additional considerations are required beyond the algorithm itself. In [2], we further discuss (1) TTJ cost modeling to determine both $\mathcal{T}_Q$ and $\mathcal{P}_Q$; (2) using TTJ with buhsy plan, including the construction of a buhsy plan from a $\mathcal{T}_Q$ and a formal analysis of TTJ performance; and (3) an extended TTJ for cyclic queries with a formal runtime analysis.

4 CORRECTNESS AND OPTIMALITY OF TTJ

We prove the correctness and the optimality gurantee of TTJ in this section. Due to the space limit, we present the correctness theorem without the proof and focus on the proof of optimality. The omitted lemmas and proofs are in [2].

Theorem 4.1 (**Correctness of TTJ**). *Evaluating an ACQ of k relations using $\mathcal{P}_Q$, which consists of $k-1$ instances of Algorithm 3.1 as the join operators and 1 instance of TTJ scan (Algorithm 3.2) for the left-most relation R_k, computes the correct query result.*

The runtime analysis of evaluating $\mathcal{P}_Q$ is done in two steps. First, we propose a general condition for any left-deep plan without cross-product for ACQ called *clean state*. Clean state specifies what tuples can be left in the input relations without breaching the $\mathcal{O}(n+r)$ evaluation time guarantee. In contrast to the common belief that input relations have to be free of dangling tuples to enable $\mathcal{O}(n+r)$ evaluation, clean state permits the existence of dangling tuples. Clean state provides a formal explanation on one reason why YA may have large dangling tuple removal costs — it spends efforts to remove more than necessary tuples. Second, we show $\mathcal{P}_Q$ reaches the clean state and the work done by TTJ between the beginning of the query evaluation and reaching the clean state (*cleaning cost*) is no more than the work done after reaching the clean state. The former takes $\mathcal{O}(n)$ and the latter takes $\mathcal{O}(n+r)$.

*Definition 2 (**clean state**).* For a left-deep plan without cross-product for ACQ, we denote the contents of R_i that satisfy the following conditions by $\widetilde{\mathbb{R}}_i$:

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

(ii) $(\mathbb{R}_i \ltimes J^*_{i+1}) \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u = \emptyset$ for internal° relations R_i and their child relations R_u; and

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

The plan reaches *clean state* if the contents of all R_i equal $\widetilde{\mathbb{R}}_i$.

Lemma 4.2. *When the left-deep plan without cross-product for ACQ is in clean state, R_k is fully reduced and free of dangling tuples.*

Theorem 4.3 (**Clean state implies optimal evaluation**). *Once the left-deep plan without cross-product is in clean state, any intermediate results generated from the plan evaluation will contribute to the final join result and the plan can be evaluated optimally.*

Comparison with full reducer and reducing semijoin program. Relations that are free from dangling tuples are in clean state. Thus, relations after F_Q are in clean state. Relations after HF_Q are in clean state as well. Leaf relations after HF_Q satisfy Condition (i) (by definition of HF_Q) and the root relation after HF_Q satisfies Condition (iii) (by Lemma 4.2 and Lemma 4 of [13]). For an internal° relation R_i, it satisfies $\mathbb{R}_i \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u = \emptyset$, which implies the satisfaction of Condition (ii). However, the state of relations after HF_Q or F_Q is stricter than what is required by clean state, i.e., more than necessary tuples are removed for 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. But, those dangling tuples are allowed to present in clean state.

Example 11. Consider a $\mathcal{T}_Q$ $R_3(x) \to R_2(x, y) \to R_1(y)$ with the following database instance: $R_3(4)$, $R_2(4, 6)$, $R_2(3, 5)$, $R_2(3, 7)$, $R_2(4, 7)$, and $R_1(7)$. Clean state only requires the removal of one tuple $R_2(4, 6)$. HF_Q removes two tuples $R_2(4, 6)$ and $R_2(3, 5)$. F_Q removes three tuples: $R_2(4, 6)$, $R_2(3, 5)$, and $R_2(3, 7)$.

Lemma 4.4. *When TTJ finishes execution, $\mathcal{P}_Q$ is in clean state.*

Lemma 4.5. *TTJ evaluates $\mathcal{P}_Q$ in $\mathcal{O}(n+r)$ once it is in clean state.*

Next, we prove the optimality guarantee of TTJ by bounding the cleaning cost. The key idea is to leverage the fact that whenever a dangling tuple is detected, some tuple has to be removed and there can be at most kn tuples removed. The cost to remove each tuple is $\mathcal{O}(1)$ under data complexity.

Theorem 4.6 (**Data complexity optimality of TTJ**). *Evaluating an ACQ of k relations using $\mathcal{P}_Q$, which consists of $k-1$ instances of Algorithm 3.1 as the join operators and 1 instance of TTJ scan (Algorithm 3.2) for the left-most relation R_k, has runtime $\mathcal{O}(n+r)$, meeting the optimality bound for ACQ in data complexity.*

Proof. By Lemma 4.4, the execution of a plan is in clean state when TTJ execution finishes. The amount of work that makes $\mathcal{P}_Q$ 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 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 `getNext()` 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 $\le \sum_{i=1}^{k-2}(k-i)\cdot n = O(k^2 n)$.

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 `deleteDT()` calls from Line 20, which is $O(kn)$.

Summing everything together, cleaning cost is $O(k^2 n)$. Since $\mathcal{P}_Q$ is clean after computing the first join result, with Lemma 4.5, the result follows. □

The combined complexity of TTJ is $O(k^2 n + kr)$, which can be further reduced to $O(nk \log k + kr)$ by imposing an additional constraint on $\mathcal{P}_Q$. We defer the details to [2].

5 EVALUATION

We compare the performance of TTJ with the baselines (§ 5.3), introduce three parameters that impact TTJ performance, and analyze them through control studies (§ 5.4). We further examine the space consumption of *ng* and the robustness of TTJ (§ 5.4).

5.1 Algorithms and Implementation

We compare TTJ with the baselines (§ 2.3) in an apples-to-apples fashion, where we implement all these methods within the same query engine built from scratch in Java. The engine architecture is similar to the architecture of recent federated database systems [12, 54]. The engine optimizes each algorithm using the same DP procedure [26] with an algorithm-specific cost model[6]. Due to the space limit, we defer the details of the cost models to [2]. The engine connects two data sources: PostgreSQL 13, which provides the estimation to the terms in the cost models, and DuckDB [50], which serves as the storage manager. All data are stored on disk.

[6]All cost models estimate the sum of intermediate result sizes

We detail the implementation of *ng* here. Suppose R_k has m children $S_1, \ldots, S_m$. Physically, *ng* is implemented as a hash table $\langle S_i, \ell_i \rangle$ where ℓ_i is a set containing $jav(t, R_k, S_i)$ for dangling tuple t from R_k detected by S_i.

We provide additional implementation details of the baselines that are not described in § 2.3. To implement YA, we introduce a k-ary physical operator *full reducer operator* that executes F_Q. The fully reduced relations, which already reside in memory, are then evaluated by HJ. PT is implemented similarly to YA with a k-ary operator for the predicate transfer phase. PT originally works on the predicate transfer graph, which contains redundant edges compared with $\mathcal{T}_Q$. Redundant edges may lead to additional unnecessary passes of Bloom filters that may negatively impact PT performance [7]. Thus, we show the results of PT on $\mathcal{T}_Q$. We use the blocked Bloom filter [48] implementation from [31].

5.2 Experimental Setup

Workload. We use three workloads: Join Ordering Benchmark (JOB) [40], TPC-H [58] (scale factor = 1), and Star Schema Benchmark (SSB) [47] (scale factor = 1). We focus on ACQs in the benchmarks, i.e., we omit cyclic queries, single-relation queries, and queries with correlated subqueries. All 113 JOB queries, 13 TPC-H queries, and all 13 SSB queries meet the criteria.

Environment. For all our experiments, we use a single machine with 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 are stored on JVM heap. Benchmarks are orchestrated by JMH [1], which includes 5 warmup forks and 10 measurement forks for each query and algorithm. Each fork contains 3 warmup and 5 measurement iterations.

5.3 Comparison with Existing Algorithms

5.3.1 Query Performance. Figure 4 compares the execution time of TTJ, YA, and PT against HJ on JOB queries. Of all 113 queries, TTJ runs faster than HJ on 112 (99%) of them. The maximum speedup is 6.8× (6.c) and the minimum speedup is 1× (6f). On average (geometric mean), TTJ is 1.8× faster than HJ. YA is faster than HJ on 47 (42%) queries. The maximum, average, and minimum speedup is 11.3× (5a), 1×, 0.3× (6f), respectively. PT is faster than HJ on 67 (59%) queries. The maximum, average, and minimum speedup is 11.5× (5a), 1.1×, 0.3× (15b), respectively. From the aggregate statistics we can see that (1) TTJ has more steady speedup than YA and PT on the entire workload: TTJ has higher average and minimum speedup than the other two algorithms; (2) YA and PT can outperform TTJ in special cases such as 5a, which returns empty results. 5a is favorable for YA and PT because the query evaluation terminates earlier than TTJ: The first semijoin movie_companies ⋉ company_type in the bottom-up pass completely removes all the tuples in movie_companies, which subsequently terminates the whole query evaluaiton. In contrast, the two relations appear as the second and the fourth relation in $\mathcal{P}_Q$, which makes TTJ perform more join computations than YA and

[7]We conducted an empirical study by comparing PT on the predicate transfer graph with the same PT on $\mathcal{T}_Q$ to verify our conclusion. Result shows PT on $\mathcal{T}_Q$ outperforms PT on the predicate transfer graph by 1× [2].

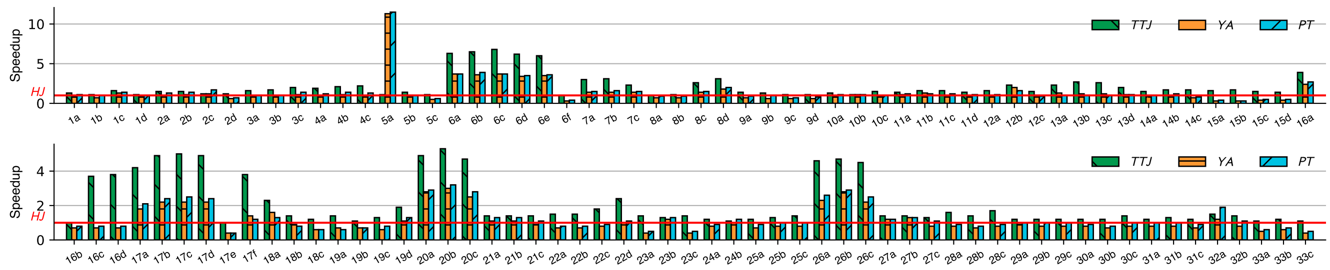

Figure 4: Speedup of TTJ, YA, PT over HJ on all 113 JOB queries

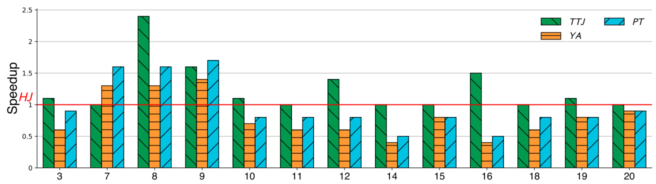

Figure 5: Speedup of TTJ, YA, PT over HJ on 13 TPC-H queries

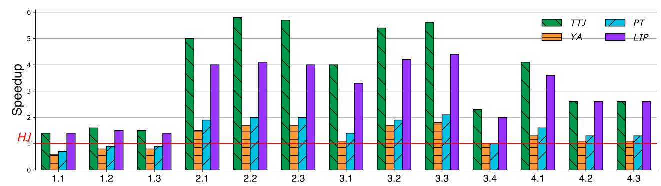

Figure 6: Speedup of TTJ, YA, PT, and LIP over HJ on all 13 SSB queries

PT before it terminates. This exemplifies the importance of join order for TTJ, which we further study in § 5.4.5.

Figure 5 shows the comparison result on TPC-H. TTJ has the maximum speedup 2.4× on Q8, the largest query with $k = 8$ in TPC-H. 2.4× is also the largest speedup among the three algorithms. Similar to its performance pattern on JOB queries, TTJ has steady speedup over the benchmarked TPC-H queries with average 1.2× compared with 0.69× from YA and 0.84× from PT. We further study a few interesting TPC-H queries in § 5.3.2.

For star schema queries, all algorithms share the identical $\mathcal{T}_Q$ and plan, where the fact table is R_k and the dimension tables are the children of R_k ordered from left to right. Figure 6 illustrates TTJ has the largest speedup, 3.2× on average, for all SSB queries and LIP comes in second with average of 2.8×. After eliminating the impact of join order and join tree, the performance difference between TTJ and LIP shows that lazily building and probing *ng* works better than proactively building and probing a set of Bloom filters. Probing Bloom filters at R_k in LIP can be viewed as performing a bottom-up pass of $\mathcal{T}_Q$. Compared with LIP, YA and PT perform an additional top-down pass of $\mathcal{T}_Q$. The potential benefit of the top-down pass performed by YA or PT can be very small because the fact table is fully or nearly fully reduced after the bottom-up pass [13] and the dangling tuples in the dimension tables will not or unlikely be matched during join evaluation. A possible performance gain from the top-down pass is from dimension table size reduction, which can speed up hash table operations. Both YA (average 1.2×) and PT (average 1.4×) are slower than LIP, indicating that the cost of performing the top-down pass of $\mathcal{T}_Q$ outweighs the potential benefit due to dimension table size reduction. PT comes the third and runs faster than YA because Bloom filter probe is faster than semijoin hash table probe.

5.3.2 *Trade-off between join time and removing dangling tuple time.*

All the join algorithms we studied strategically allocate runtime between performing joins and removing dangling tuples. On one end of the spectrum, HJ spends all of its runtime performing joins. On the other end of the spectrum, YA, PT, and LIP spend most of its runtime removing dangling tuples. PT spends less than YA due to the efficiency of Bloom filters. LIP further reduces dangling tuple removal time on star schema queries by eliminating the top-down pass of $\mathcal{T}_Q$. Due to the laziness nature of TTJ, it aims to stay closer to the HJ side by spending less of its runtime on removing dangling tuples and more time on computing joins. Figure 7 illustrates the patterns by showing the runtime breakdown on TPC-H queries [8]. The figure shows that each algorithm's overall performance largely depends on its *dominate time*, i.e., join time for TTJ and dangling tuple removal time for YA and PT.

YA and PT are performant when the full reducer can be executed quickly. Consider Q7: A fragment of YA join tree is a chain orders → lineitem → supplier → nation. The first semijoin supplier ⋉ nation already removes more than 90% of tuples from supplier because |nation| = 1. The largely reduced supplier speeds up the subsequent semijoin lineitem ⋉ supplier and starts a chain reaction on the remaining semijoins. As a result, YA removes close to 100% of the tuples of the input relations (Figure 8) in a small amount of time (Figure 7). PT shares the same join tree as YA and has a similar behavior. On the flip side, YA and PT face challenges when the full reducer executes slowly. A typical example is star schema queries. Figure 9 shows the fraction of input relations tuples removed on SSB. From the figure we see that YA and PT remove almost identical number of dangling tuples as LIP but have much lower speedup (Figure 6). This shows that the top-down pass of $\mathcal{T}_Q$ that YA and PT perform on star schema queries not only incurs additional execution cost but also can hardly reduce dimension table size.

TTJ performs better when its join time is smaller than the dangling tuple removal time of YA and PT. Join time is usually small

[8] Due to the space limit, we defer the runtime breakdown of LIP on SSB to [2], which illustrates LIP spends less time on dangling tuple removal than YA and PT but more than TTJ.

if a large number of dangling tuples can be removed. Thus, intuitively, TTJ is good if the small amount of dangling tuple removal time spent by TTJ can remove a huge number of dangling tuples. In Q8, a typical example that TTJ greatly outperforms YA and PT, TTJ removes 91% of the dangling tuples removed by YA or PT, while using only 22% of YA's and 27% of PT's dangling tuple removal time. However, the quantity of dangling tuples removed alone is not a decisive factor on explaining the performance of TTJ. For example, in Q15, TTJ spends a neglible amount of time removing the same number of dangling tuples as YA and PT (99% of input tuples) but unlike Q8, the join time is not significantly reduced. As a result, TTJ does not considerably outperform YA and PT. Such observation indicates that the *quality* of dangling tuples removed also matters. A dangling tuple has high quality if removing it can substantially reduce the join time. Directly measuring dangling tuple quality is non-trivial; instead, we use two parameters to measure the effectiveness of the actions to remove certain groups of dangling tuples. The more effectiveness the actions are, the higher quality the removed dangling tuples have.

Duplicate ratio α. *ng* contains all the unique *jav*s of the dangling tuples in R_k. The action taken by TTJ related to *ng* contains two steps: (1) If R_k is the guilty relation, *jav* is computed and put into *ng*; (2) Future tuples from R_k are filtered out if their *jav* appear in *ng*. We focus on the filtering step of the action. To measure its effectivness, we can divide the dangling tuples of R_k into two sets: Set A contains dangling tuples that can be filtered out by *ng* and set B contains the rest of the dangling tuples. We define $\alpha = \frac{|A|}{|A|+|B|}$, which is the fraction of tuples in the dangling tuples of R_k that can be filtered out by *ng*. The larger α is, the more dangling tuples can be filtered out by *ng*. For example, 99% of the tuples in lineitem (R_k of Q8) is dangling. Its α is 96%.

Modified Semijoin Selectivity θ. On detecting dangling tuples, `deleteDT()` is called. If the guilty relation is an internal° relation, a tuple is removed from its hash table. We denote the action of removing dangling tuples from the hash tables as *rm*. θ_R measures the fraction of tuples from an internal° relation R that will no longer participate join once the dangling tuples from all of its child relations are removed. The larger θ_R is, the more effective *rm* is. For example, in Q8, customer has the highest θ 6.6%. We provide the formal definition of θ in [2] and give an example in § 5.4.2.

With the concept of quality, we can say that TTJ is fast when it can remove a large number of high quality dangling tuples within a small amount of dangling tuple removal time. We introduce a third parameter, *backjumping distance*, which determines how fast a dangling tuple can be removed.

Backjumping distance b_{ij}. When join fails, TTJ backjumps to the guilty relation via `deleteDT()` calls. We call the action bj. b_{ij} denotes the number of relations between the detection relation R_i (excluding) and the guilty relation R_j (including) for a join failure. The larger b_{ij} is, the quicker the dangling tuple from the guilty relation can be removed. Join time is also reduced because backjumped relations (relations appear between the detection and the guilty relations in $\mathcal{P}_Q$) will no longer be probed until a new join result is produced by the guilty relaiton. In Q8, the largest b_{ij} is 4.

5.4 Detailed Analysis of TTJ

We perform control studies on the parameters introduced in § 5.3.2 to measure the effectivness of the corresponding TTJ actions.

Result Summary. Query and database instance can lead to a large number of high quality dangling tuple removal if (1) duplicate ratio $\alpha > 50\%$ (§ 5.4.1); (2) modified semijoin selectivity $\theta > 2\%$ (§ 5.4.2). Backjumping is more effective when $b_{ij} > 4$ ($k > 5$). Furthermore, we show that: (1) no-good list takes small spaces on the benchmark workloads (§ 5.4.4); (2) join order has a large impact on TTJ performance, but even with a suboptimal join order, TTJ can still match or outperform HJ.

5.4.1 *Impact of α.*

Consider the following query

$$Q = T(a,b) \bowtie R(a) \bowtie S(b) \tag{3}$$

T is the root of $\mathcal{T}_Q$ and $\mathcal{P}_Q = [T, R, S]$. Let all tuples in T be dangling due to S, i.e., $T \ltimes R = T$ and $T \ltimes S = \emptyset$. $|A| + |B| = |T|$ and $|A| = \alpha|T|$. Column $T.a$ and $R.a$ contain the numbers from 1 to $|T|$. For $T.b$, we first put $|B|$ unique values; then, we append additional $|A|$ values that are sampled from the unique values uniformly at random. We fill in $S.b$ with values that are not in $T.b$. We shuffle all the rows of all the relations at the end. All three relations have equal size of 10 million tuples.

Result Analysis. Figure 10 shows the fraction of *ng* build and probe time over the overall runtime with different α. The left-most bar shows that *ng* operations take 8% of runtime when $\alpha = 0\%$, i.e., all dangling tuples in T have unique *jav*s. The fraction stays between 8% and 10% when $\alpha \leq 50\%$. Once $\alpha > 50\%$, the fraction starts a steady drop. The right-most bar ($\alpha = 100\%$ [9]) has 2% fraction and the lowest execution time overall. In general, the larger α, the less time *ng* operations takes, and the better TTJ performs.

5.4.2 *Impact of θ.*

Condier the following micro-benchmark query:

$$R(a,c) \bowtie U(c,e) \bowtie V(c,d) \bowtie T(d,g) \bowtie W(d,f) \tag{4}$$

$\mathcal{P}_Q = [R, U, V, T, W]$. $\mathcal{T}_Q$ starts R as the root and has a chain $R \rightarrow U \rightarrow V$. Both T and W are children of V. R has two tuples $(1,2), (1,4)$. U has tuples $(2,1), (2,2), \ldots, (2, \theta|U|), (3, \theta|U|+1), \ldots, (3, |U|-1), (4,4)$, where θ is defined below. V has two tuples $(2,3), (4,4)$. T has two tuples $(3,1)$,$(4,1)$. W has one tuple $(4,5)$. The query result set is $\{(1,4,4,4,1,5)\}$.

We define θ on U as $\theta_U = \frac{|(U \ltimes R) \ltimes (\overline{V \ltimes \widetilde{V}})|}{|U|}$. $\widetilde{V}$ is V in clean state. In words, θ is the fraction of tuples in U that are joinable with R and joinable with the dangling tuples from V. We compare TTJ^{bj} with TTJ^{bj+rm}. TTJ^{bj} only enables bj and disables *ng* and *rm*. TTJ^{bj+rm} enables bj and *rm*, which removes the dangling tuples from the hash tables in addition to backjumping. We fix $|U| = 1$ million.

Result Analysis. Figure 11 shows that (1) the larger θ is, the more beneficial removing dangling tuples from the hash tables becomes; (2) in our implementation, it is always beneficial to remove dangling tuples from the hash tables: When $\theta = 0\%$ [10], removing dangling tuples will not reduce subsequent join computations, but in such case, TTJ^{bj+rm} and TTJ^{bj} still have matching performance.

[9]Technically, $\alpha = 99.9\%$ because $|B| \geq 1$, i.e., there has to be at least one tuple in B so that tuples in A can be filtered out by *ng*.

[10]Technically, $\theta > 0\%$ because in our micro-benchmark, U at least has $(2,1)$ and θ is at least one over 1 million.

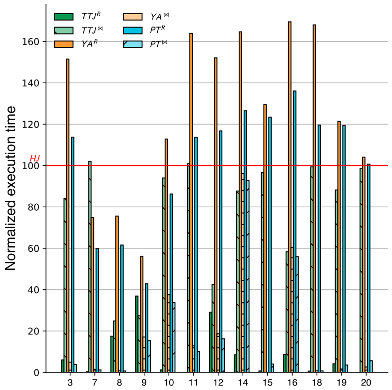

Figure 7: Breakdown of TTJ, YA, and PT execution time into dangling tuples removal (e.g., TTJ^R) and join (e.g., $\mathsf{TTJ}^{\bowtie}$) on TPC-H

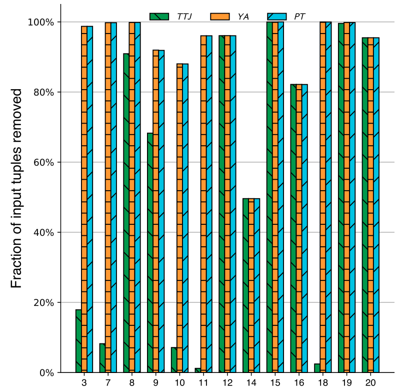

Figure 8: Fraction of tuples removed from the input relations by TTJ, YA, and PT on TPC-H

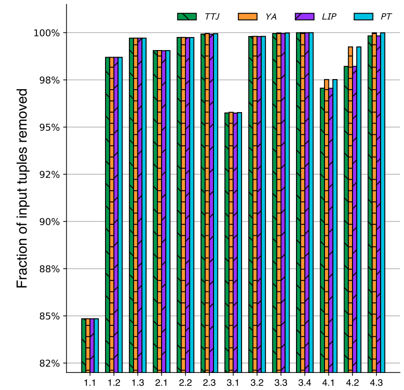

Figure 9: Fraction of tuples removed from the input relations by TTJ, YA, LIP, and PT on SSB

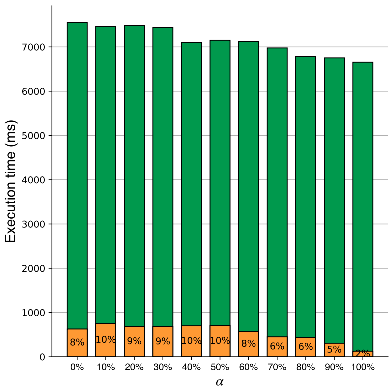

Figure 10: Execution time and profile percentage of runtime spent on building and probing ng across different α on mini-benchmark Query (3)

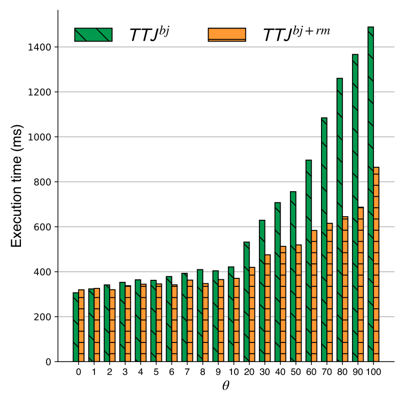

Figure 11: Execution time between TTJ^{bj} and TTJ^{bj+rm} for different θ of mini-benchmark Query (4)

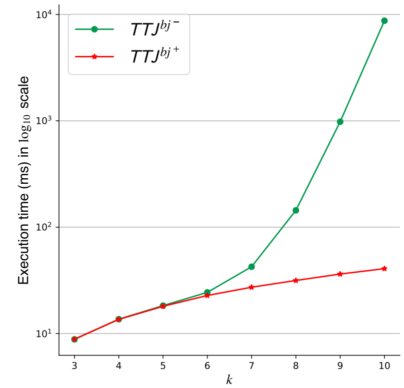

Figure 12: Execution time between TTJ^{bj^-} ($b_{ij} = 1$) and TTJ^{bj^+} ($b_{ij} = k - 1$) for different number of input relations k of mini-benchmark Query (5)

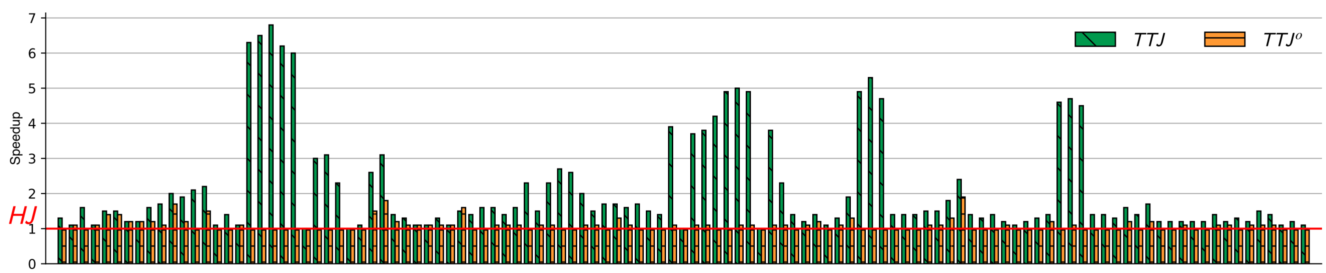

Figure 13: Performance comparison among TTJ on TTJ order (TTJ in the figure), TTJ on HJ order (TTJ^o in the figure), and HJ on HJ order on all 113 JOB queries

To explain the performance difference, first consider TTJ^{bj}, where it does not remove dangling tuples. The evaluation starts with $U(2,1)$ and does not fail until $W(d,f)$. Then, `deleteDT()` resets the evaluation flow to V. $V(2,3)$ is not removed. No more matching tuples is left from V given jav $(c : 2)$. `deleteDT()` further sets the evaluation flow to U and moves on to $U(2,2)$, which is joinable with $R(1,2)$. Since $V(2,3)$ still presents, the join result $(1,2,2,3)$ will eventually try W and fail again. `deleteDT()` brings the evaluation flow back to V. Since no more tuples are joinable with $U(2,2)$, `deleteDT()` resets the flow back to U. Then, $U(2,3)$ is returned. The same process repeats $\theta|U|$ times in total. In TTJ^{bj+rm} evaluation, $V(2,3)$ is deleted when `deleteDT()` first resets the evaluation flow to V. The evaluation will finish much earlier because $U(2,2)$ will be removed immediately once it probes into V, and the same happens to $U(2,3),\dots,U(2,\theta|U|)$.

5.4.3 Impact of b_{ij}. For this study, we only enable the backjumping action of TTJ (TTJ^{bj}). Consider the following query (for simplicity, we replace ⋈ with comma between relations):

$$Q = R_1(a_1,\dots,a_k), R_2(a_2,a_3),\dots,R_{k-1}(a_{k-1},a_k), R_k(a_k) \quad (5)$$

The database instance is as follows: $R_1(a_1,\dots,a_k)$ has two tuples $(1,2,3,\dots,k-1,k)$ and $(1,3,4,5,\dots,k,k+1)$. $R_i(a_i,a_j)$ has $n-1$ copies of (i,j) and a tuple $(i+1,j+1)$. $R_k(a_k)$ has one tuple $(k+1)$. Query (5) has only one join result $(1,3,4,5,\dots,k+1)$.

We run TTJ^{bj} on two $\mathcal{T}_Q$: $\mathcal{T}_Q^-$ is a chain shape: $R_1 \to R_2 \to \dots \to R_k$. $\mathcal{T}_Q^+$ is a star shape: R_1 is the root and the rest of the relations are its children ordered from left to right. We denote TTJ^{bj} on $\mathcal{T}_Q^-$ as TTJ^{bj^-} and on $\mathcal{T}_Q^+$ as TTJ^{bj^+}. TTJ^{bj^-} has the characteristic that every guilty relation S is immediately before the detection relation R for any join failure, i.e., $b_{ij}=1$. TTJ^{bj^+} has only one join failure, which happens when the tuple $(k-1,k)$ of R_{k-1} joins with R_k. After the join failure, TTJ^{bj^+} resets the execution flow to R_1 and starts to compute the final join result. Thus, $b_{ij}=k-1$. TTJ^{bj^-} produces $(n-2)\sum_{j=0}^{k-3}(n-1)^j = O(kn^k)$ more dangling intermediate results than TTJ^{bf^+} [2]. We fix $n=10$ and vary k.

Result Analysis. Figure 12 shows that the performance between TTJ^{bj^+} and TTJ^{bj^-} begins to diverge when $k=6$ ($b_{ij}=5$) where TTJ^{bj^-} produces 6560 more dangling tuples than TTJ^{bj^+} does. After that, we see the execution time of TTJ^{bj^-} grows exponentially whereas TTJ^{bj^+} grows logrithmically. The result indicates that the backjumping distance impacts the number of dangling tuples that can be avoided by TTJ, thereby affecting TTJ performance.

Table 2: Number of *javs* stored in *ng*. In parenthesis, we list memory percentage consumption taken by *ng* with respect to total query evaluation memory consumption

Bench.	min	max	avg.
JOB	12 (0%)	4051176 (6%)	908226 (0.3%)
TPC-H	24 (0%)	1470901 (4.2%)	176716 (0.6%)
SSB	2041 (0%)	201343 (1.9%)	65223 (0.4%)

5.4.4 Space Consumption of ng. Table 2 shows the space taken by *ng* on the benchmark queries. Despite of the relatively large *ng* size, the memory footprint is negligible, e.g., at most 6% of total memory consumption. The main reason is that *ng* only stores *javs* (a few integers), which are tiny compared with other memory consumption, e.g., loading relations into memory.

5.4.5 Robustness against Poor Plans. In this experiment, we study whether TTJ performance is robust against poor plans. We compare three setups: (1) TTJ on HJ order (we call it TTJ^o); (2) TTJ on TTJ order; and (3) HJ on HJ order. We consider HJ order as a poor plan because the order is not specific optimized for TTJ. Figure 13 shows that compared with TTJ, the number of queries that TTJ^o outperforms HJ is smaller (105 vs. 112) and the average speedup goes down (1.1× vs. 1.8×). This result shows that in general, optimizing TTJ specifically can lead to much larger performance gain compared with treating TTJ as HJ. Nevertheless, TTJ still matches or outperforms HJ on HJ order.

6 DISCUSSION AND RELATED WORK

We organize the related work in four categories. (1) *CSP.* The equivalence between CQ evaluation and CSP is established by [16, 39]. TreeTracker in [10] solves a CSP for one solution without preprocessing the CSP. TTJ extends TreeTracker into query evaluation by (a) returning all possible solutions; (b) blending the ideas from TreeTracker into physical operators in a query plan. (2) *Semijoin reduction.* An intensive research has been done on using semijoin to improve query evaluation speed [13, 14, 18, 37, 41, 60, 61, 67]. TTJ achieves a similar effect (clean state) as performing semijoin reduction without explicitly using semijoins. (3) *SIP* `deleteDT()` of TTJ takes the form of SIP [9, 22, 23, 25, 30, 32, 33, 36, 43, 45, 49, 53, 55, 70, 71]. TTJ is different from the prior approaches in one or more of the following aspects: (a) TTJ does not introduce any preprocessing steps; (b) TTJ does not use Bloom filters, bitmaps, or semijoins; and (c) TTJ provides optimality guarantee. (4) *Worst-Case Optimal Join (WCOJ) algorithms.* A related line of work is to implement WCOJ algorithms efficiently [3, 7, 24, 35, 44, 64, 65]. TTJ is orthogonal to such direction as TTJ focuses on ACQ evaluation (§ 2.2). We designed an extended TTJ [2] that works for cyclic CQ evaluation. Comparing to WCOJ algorithms, which commonly use multi-way join operators, the extended TTJ uses binary physical operators in iterator interface.

7 LIMITATIONS AND FUTURE WORK

We propose the first join algorithm that incorporates backjumping and no-good into query evaluation. Gaps remain when consider TTJ with additional requirements from both practical and theoretical aspects, which we discuss next. *Practical aspects.* (1) We focus on estimating the logical cost in our cost model for TTJ. Future extension to the model can include physical cost coefficients such as *ng* probing cost, hash table probing cost, and tuple deletion cost, and so on; (2) we present TTJ using the tuple-based iterator interface. Extending TTJ to work with vectorization has one challenge: Batch processing introduces an additional trade-off because it reduces the number of recursion calls, but potentially loses the opportunity for detecting and deleting dangling tuples; (3) TTJ assumes demand-driven pipelining and requires additional extension to work with asychronous processing; and (4) TTJ uses *ng* only on the left-most relation R_k. Whether using *ng* on the other relations requires further assessment on the *ng* probing cost versus the potential additional dangling tuple removal. *Theoretical aspects.* (1) The combined complexity of TTJ can be improved because it has an additional $\log k$ term compared with the complexity of YA; (2) the extended TTJ for cyclic queries does not have the same complexity as WCOJ algorithms do, which requires further exploration.

REFERENCES

[1] [n.d.]. Java Microbenchmark Harness (JMH). https://github.com/openjdk/jmh
[2] [n.d.]. Omitted due to Double Anonymous Requirement.

[3] 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

[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] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-Based Pruning. *Proc. ACM Manag. Data* 1, 2, Article 167 (jun 2023), 26 pages. https://doi.org/10.1145/3589312

[7] 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.

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

[9] 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

[10] 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.

[11] 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

[12] 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

[13] 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

[14] Philip A Bernstein and Nathan Goodman. 1981. Power of Natural Semijoins. *SIAM J. Comput.* 10, 4 (1981), 751–771.

[15] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[16] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/800105.803397

[17] 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.

[18] 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

[19] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. 1990. Abstract Machine for $\mathcal{LDL}$. In *EDBT*.

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

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

[22] 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

[23] 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

[24] 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

[25] 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.

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

[27] John Gaschnig. 1979. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD dissertation. Deptment of Computer Science, Carnegie Mellon University.

[28] 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

[29] 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

[30] 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

[31] Thomas Mueller Graf and Daniel Lemire. 2022. Binary Fuse Filters: Fast and Smaller Than Xor Filters. *ACM J. Exp. Algorithmics* 27, Article 1.5 (mar 2022), 15 pages. https://doi.org/10.1145/3510449

[32] 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.

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

[34] Roberto J. Bayardo Jr and Daniel P. Miranker. 1996. Processing queries for first-few answers. In *Proceedings of the fifth international conference on Information and knowledge management*. 45–52.

[35] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible Caching in Trie Joins. *EDBT* (2017). https://openproceedings.org/2017/conf/edbt/paper-131.pdf

[36] 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

[37] H. Kang and N. Roussopoulos. 1991. A Pipeline N-Way Join Algorithm Based on the 2-Way Semijoin Program. *IEEE Transactions on Knowledge & Data Engineering* 3, 04 (oct 1991), 486–495. https://doi.org/10.1109/69.109109

[38] 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

[39] 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

[40] 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

[41] Zhe Li and Kenneth A. Ross. 1995. PERF Join: An Alternative to Two-Way Semijoin and Bloomjoin. In *Proceedings of the Fourth International Conference on Information and Knowledge Management* (Baltimore, Maryland, USA) *(CIKM '95)*. Association for Computing Machinery, New York, NY, USA, 137–144. https://doi.org/10.1145/221270.221360

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

[43] 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

[44] 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

[45] 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

[46] 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
[47] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–252.
[48] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM J. Exp. Algorithmics* 14, Article 4 (jan 2010), 18 pages. https://doi.org/10.1145/1498698.1594230
[49] 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
[50] 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
[51] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). McGraw-Hill.
[52] Kenneth Rosen. 2011. *Discrete Mathematics and Its Applications* (7th ed.). McGraw Hill.
[53] 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
[54] 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
[55] 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.
[56] Abraham Silberschatz, Henry F. Korth, and Shashank Sudarshan. 2019. *Database System Concepts* (7th ed.). McGraw-Hill New York.
[57] 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
[58] 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.
[59] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. (2020), 2659–2665. https://doi.org/10.1145/3318464.3383132
[60] Jeffrey D. Ullman. 1989. *Principles of Database and Knowledge-Base Systems Vol. 2: The New Technologies* (first ed.). Computer Science Press, USA.
[61] Patrick Valduriez and Georges Gardarin. 1984. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Trans. Database Syst.* 9, 1 (mar 1984), 133–161. https://doi.org/10.1145/348.318590
[62] 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
[63] 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
[64] 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
[65] 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
[66] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. *Conference on Innovative Data Systems Research (CIDR)* (2024). arXiv:2307.15255 [cs.DB]
[67] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*, Vol. 81. 82–94.
[68] 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
[69] 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
[70] Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proc. VLDB Endow.* 16, 11 (jul 2023), 2962–2975. https://doi.org/10.14778/3611479.3611501
[71] 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