Treetracker Join: A Composable Physical Operator that Simultaneously Computes Join and Semijoin

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Yannakakis's seminal algorithm (YA) for the optimal execution of k-way acyclic conjunctive queries (ACQ) requires executing 2 rounds of $k-1$ semijoins to remove dangling tuples, followed by executing the joins. This paper presents the TreeTracker Join algorithm (TTJ) that removes the explicit semijoins. TTJ simultaneously implements the functionality of a join operator and a semijoin operator. In essence, TTJ merges the behavior of two logical operators into a single physical operator. We prove that by composing $k-1$ TTJ operator instances, the result of a k-way acyclic conjunctive query can be computed in optimal data complexity time, $\mathcal{O}(n+r)$, where $\mathcal{O}(n)$ and $\mathcal{O}(r)$, are the size of the input and output. No additional operators are needed. A distinctive feature of TTJ is that it detects dangling tuples during the execution of a binary join and removes them, rather than removing dangling tuples and then performing joins on the result.

An emperical evaluation of TTJ on two commonly used benchmarks shows that in most cases TTJ is faster than YA, an improvement to YA as well as two representative filter methods.

2012 ACM Subject Classification Information systems → Join algorithms; Information systems → Query operators

Keywords and phrases Optimal Join Algorithms, Acyclic Conjunctive Queries

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Improving join performance is perennially important to the database community. Formal studies of queries with multiple joins commonly use conjunctive queries (CQ) as a model. Until the development of worst-case optimal join (WCOJ) algorithms, the evaluation of queries comprising multiple joins focused on the composition of the unary and binary operators of the relational algebra[27]. In 1981 Yannakakis published his seminal optimal algorithm for the special case of acyclic conjunctive queries (ACQs)[66]. Hereafter we will write Yannakakis's algorithm as YA.

YA comprises the logical composition of joins and semijoins. The practical benefit of YA is situational. Even though it has been established that an overwhelming majority of relational queries in real-world applications are acyclic [24] and YA is optimal it is rare that YA is the basis for the most performant query plan.

YA uses a logical composition of relational joins and semijoins in two phases, the semijoin reducation phase and the join phase. The semijoin reduction phase, called *full reducer* $F_{\mathcal{Q}}$, comprises two passes of $k-1$ semijoins totaling $2k-2$ semijoins [12]. The first pass, called *reducing semijoin program* $HF_{\mathcal{Q}}$, follows a bottom-up traversal of the query's join tree $\mathcal{T}_{\mathcal{Q}}$, removing dangling tuples by evaluating a semijoin for each vertex: $R_p \ltimes R_c$, where R_c is a relation associated with a node in $\mathcal{T}_{\mathcal{Q}}$ (*child relation*) and R_p is the relation associated with the parent of the node (*parent relation*) [12]. The second pass is the same except it traverses the path in the reverse direction. The two passes remove all dangling tuples from the input relations. In otherwords a tuple remains as an input argument if and only if it

appears in the final result. The expression "fully semijoin reduced" is used to describe this state. We use R_i' to indicate the resulting relations after $HF_{\mathcal{Q}}$ and R_i^* to represent the fully semijoin reduced relations. The join phase then enumerates the final results. It is commonly understood that only one of the semijoin passes in YA is necessary to obtain the optimal complexity result [62, 31] [1]. Hereafter the single pass version of YA will be written as YA$^+$. The YA is foundational for a preponderance of research and practice that replaces the goal of achieving algorithmically optimal query plans with a cost assessment of each opportunity to apply a semijoin and based on the trade-off estimated by the cost assessment an operation to remove dangling tuples may or may not be included. The compact size and speed advantage of Bloom filters is sufficient that methods that use Bloom filters [14] and approximate semijoins in the form of sideways information passing (SIP) dominate [68, 32, 35, 33, 29, 44, 22, 52, 54, 48, 23, 26, 3].

More recently WCOJs have been developed. One goal, similar to the efforts inspired by YA, is to eliminate the introduction of additional operators and their concomitant overhead. Much of this research has met with success. However, WCOJs commonly compute a k-way join using a single operator of k inputs for all possible k. Recent research on WCOJs includes determing how a k-way join operator may be introduced into query systems that historically have only been required to optimize queries composed of unary and binary relational operators [29, 25, 64].

In this paper, we offer an alternative approach, Treetrack Join (TTJ). TTJ combines the implementation of a semijoin and join into one physical operator. The conceptual foundation of the approach is captured by the Datalog program in **Example 1**. A goal of presenting TTJ in logical form is to make clear the algorithm takes two relations as inputs and produces a join result as output. This supports a claim that will be detailed further, that TTJ can be integrated with conventional join algorithms in a query plan and is downwardly compatible with many RDBMS query systems. To implement the deletion of dangling tuples TTJ has an additional input and output. These are the atoms, $DDT_{\bowtie_2}()$ and $DDT_{\bowtie_0}()$ appearing in **Rules** (2) **and** (4). A short-intuitive explanation that TTJ is downwardly compatable is that semijoin reduction is an optimization and simply ignoring $DDT_{\bowtie_2}()$ and $DDT_{\bowtie_0}()$ does not impact the correctness of the final result.

The definition of the TTJ algorithm follows in Section 4. Proofs of correctness, complexity and emperical results appear in Sections 5–7.

▶ **Example 1.** Consider the query $\mathcal{Q} = A(x) \bowtie_0 B(x,y,z) \bowtie_1 C(y,z) \bowtie_2 D(y)$. The following Datalog program models the logical behavior of $\bowtie_1$ when three instances of TTJ are used to evaluate the query.

$$C'(y,z) \leftarrow C(y,z) \tag{1}$$

$$C'(y,z) \leftarrow C'(y,z) - DDT_{\bowtie_2}(y,z) \tag{2}$$

$$Join_1(x,y,z) \leftarrow Join_0(x,y,z), C'(y,z) \tag{3}$$

$$DDT_{\bowtie_0}(x,y,z) \leftarrow Join_0(x,y,z), C(y,z) - Join_1(x,y,z) \tag{4}$$

Queries cannot have side effects on base relations. Rule (1) copies relation C into a data structure internal and strictly local to an operator instance, enabling the removal of *dangling tuples* from further consideration, (2). Rule (3) represents the desired join computed

[1] Neither Internet search nor soliciting people active in this research area has identified a published proof. For completeness we include a proof in Appendix J.

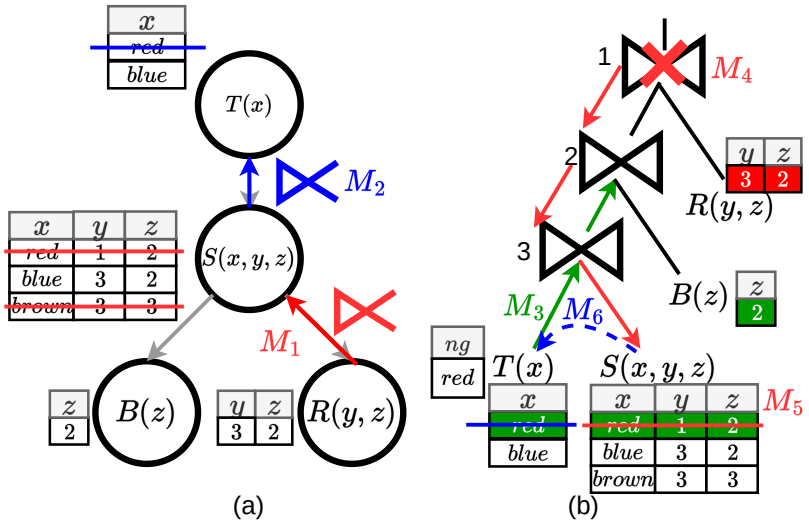

Figure 1 Illustration of the identification and removal of dangling tuples by (a) Yannakakis's algorithm (YA) its improvements as (YA$^+$); and (b) TTJ on a left-deep query plan. The labels M_i indicate algorithm execution moments referenced in Example 2 and elsewhere in the paper.

with the reduced version of C, C'. $DDT_{\bowtie_2}()$ in Rule (2) represents a subset of C such that all tuples in $DDT_{\bowtie_2}()$ are dangling tuples. The result of $Join_1$ is computed by Rule (3) without considering the removed dangling tuples. $Join_0$ in Rule (3) is the result from $A \bowtie B$. The negation in Rule (3) models determining dangling tuples by set difference. Notice the identification of a dangling tuple is not done in the same TTJ instance where the dangling tuple is represented and must be deleted. Thus the formal signature of TTJ comprises three inputs and two outputs.

As one considers the evaluation of a full query plan based on Example 1 one may anticipate that on each cycle Rule (3) adds results to $Join_1$ then on the next cycle Rule (4) determines additional dangling tuples and that these are removed from B' where B' is analogous to C' in a Datalog representation of $\bowtie_0$. Each cycle of evaluation the number of dangling tuples is greater than or equal to the number of dangling tuples processed by the previous cycle. We will show that at fixed point the number of dangling tuples deleted by TTJ can be less than the number of dangling tuples deleted by YA$^+$.

In summary, this paper makes the following contributions:

1. We design a physical join operator TTJ that computes both semijoins and joins at the same time (Section 4).
2. We prove TTJ is correct and runs optimally in data complexity for ACQ (Sections 5 and 6).
3. We define a general condition call we call *clean state* that enables optimal evaluation of an ACQ while permitting the existence of dangling tuples and show that when TTJ achieves clean state it may have removed fewer dangling tuples than either YA or YA$^+$ (Section 6.1).
4. We present empirical evidence by comparing TTJ with five baseline algorithms on three benchmarks and complete our argument that combining two logical operations into one can provide both formal guarantee and good empirical performance (Section 7).

2 Running Example

To help explain TTJ we name a particular point and state of execution a *join failure* or *join failure event*. In English and the context of simple nested loops join this is when a tuple in the outer loop has been compared to all the tuples in the relation being processed by the inner loop and the tuple from the outer loop has not joined with any tuples tested by the inner loop. Thus the tuple in the outer loop is a dangling tuple. Clarity is gained by adopting the expression that the tuple has experienced *join failure*. A formal definition of join failure is lengthy. More formal than the English definition above but incomplete the key property is, given $R \bowtie S$, a join failure event represents the moment in execution that it is determing that tuple $t \in R$ is a member of the relation $R \,\overline{\ltimes}\, S$. It remains to formaly define *event* and *moment in execution*. When used in context we believe the meaning of these terms is self-evident.

In a multi-way join evaluation, as we will now illustrate in a physical model that R can represent intermediate join results from the previous binary join computation. Unlike YA, TTJ starts join evaluation immediately. Dangling tuples are identified by monitoring for join failure and the dangling tuples are deleted as they are identified. Thus, TTJ is a join algorithm augmented to incrementally delete dangling tuples as they are identified. Hence TTJ avoids equality tests that are executed when evaluating a semijoin and then again when evaluating a join. When query evaluation terminates the internal data structures of the TTJ operator instances contain a superset approximation to the reduced arguments created by the single semijoin pass of the YA, i.e., YA$^+$. This means that TTJ may delete fewer dangling tuples than the YA or YA$^+$ yet remains data complexity optimal. See Corollary 15.

The reader may find that the programatic expression of the TTJ operator is simple when presented in isolation in English or Datalog. However since dangling tuples are often identified in one TTJ operator instance but must then be deleted from a data structure local to another operator instance the query optimizer must maintain certain constraints and represent certain consequences in the query plan. These are not simple. The query optimizer must create a certain graph representation of the query and limit plans to traversals of that graph that maintain the constraints.(See Definition 6 and Corollary 7). For an ACQ, the graph is a join tree. Edges in the traversal of the join tree determine the identification of the relation whose tuple caused a join failure and the communication path that terminates at the operator instance containing the dangling tuple, thus identifying and enabling its deletion.

▶ **Example 2.** 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. The illustrations show how dangling tuples are identified and removed by YA (and YA$^+$) and TTJ enable optimal evaluation.

(a) shows the part of the bottom-up semijoin pass in YA and YA$^+$ and highlights both of the algorithms remove more dangling tuples than TTJ in a different way. Both YA and YA$^+$ execute a sequence of semijoins prior to starting joins: At M_1, $S' = S \ltimes R$; both $S(red, 1, 2)$ and $S(brown, 3, 3)$ are removed. $S(brown, 3, 3)$ is not removed in TTJ because $x = brown$ does not match with any possible assignment to x in T. Then, at M_2, $T \ltimes S'$ and $T(red)$ is removed. Unlike TTJ that removes dangling tuples while performing join, YA removes all dangling tuples before join starts. YA$^+$ allows the existence of some dangling tuples by omitting the top-down pass but, as shown in this example, it still removes more dangling tuples than TTJ.

(b) illustrates how TTJ evaluates the same query as (a) but on a left-deep query plan using demand-driven pipelining. TTJ takes the operator form, which is 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 $\ltimes_1$'s `getNext()`, which calls its left child $\ltimes_2$'s `getNext()`, and such pattern repeats until the left most relation T's `getNext()` is called and returns $T(red)$ (M_3). $\ltimes_3$ 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 $\ltimes_2$. 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 $\ltimes_1$. No tuples from $\mathcal{H}_R$ join with $(red, 1, 2)$ (M_4); hence, join fails at R and R is the detection relation. TTJ makes additional method calls to reset the evaluation flow to S, the guilty relation, because S is the parent of R in $\mathcal{T}_\mathcal{Q}$. Subsequently, $S(red, 1, 2)$ is removed from $\mathcal{H}_S$ (M_5), 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_8). The no-good list will be used in future steps to filter out dangling tuples from T. From this example, we see that TTJ, like other join operator implementations, takes in two input relations and produce join result explicitly. However, implicitly as part of the join computation, it also identifies dangling tuples from some other relations and send deletion message and takes deletion message from some other TTJ operator to remove dangling tuples. By the end of the evaluation, dangling tuples are sufficiently removed and is a subset of the tuples removed by the semijoin passes of YA and YA$^+$.

3 Preliminaries

We examine the relevant background concerning the evaluation of acyclic conjunctive queries, present baseline algorithms, and introduce additional definitions used in this paper.

3.1 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:

$$\mathcal{Q}(\boldsymbol{a}) = R_1(\boldsymbol{a_1}) \bowtie R_2(\boldsymbol{a_2}) \bowtie \ldots \bowtie R_k(\boldsymbol{a_k}) \tag{5}$$

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

Query graph. The literature contains a number of different graph representations of $\mathcal{Q}$. The most common is the hypergraph [28, 45]. To better emphasize that TTJ leverages the connection between query evaluation and the constraint satisfaction problem (CSP)[20] , we use an equivalent alternative, *query graph* [16] (also known as *join graph* [65][2], *dual constraint graph* [20], or *complete intersection graph* [41]). The *query graph* of $\mathcal{Q}$ 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. $\mathcal{Q}$ is *acyclic* if its query graph contains a spanning tree called *join tree* $\mathcal{T}_\mathcal{Q}$, which satisfies the *connectedness property* [10, 20]: 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 $\mathcal{Q}$ is a full acyclic

[2] Join graph is defined in database theory and constraint satifaction problem with a slightly different definition: a spanning subgraph of query graph that satisfies the connectedness property [20, 41].

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 a tree is guaranteed to be a join tree [41]. 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 $\mathcal{T}_\mathcal{Q}$ 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 (5). The plan is a *left-deep query plan*, or *left-deep plan*, if the right child of every join operator is a leaf node [50]. For example, $((T \bowtie S) \bowtie B) \bowtie R$ in Figure 4 (c) is a left-deep plan. Due to the limited space this paper only discusses left-deep plan. Appendix E extends these results to bushy plans. As shorthand [63] we represent a left-deep plan, labeled from bottom to top, $(\dots((R_k \bowtie R_{k-1}) \bowtie R_{k-2})\dots) \bowtie R_1$ as $[R_k, R_{k-1}, \dots, R_1]$.

Physical Operators. Operators in the query plan of $\mathcal{Q}$ are physical operators, commonly implemented in an iterator interface [27] 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 [55]. Introducing additional methods, such as the `deleteDT()` method of TTJ, to the interface requires only minor adjustments to the current physical operators. A sole default implementation of the newly introduced methods is adequate for the current physical operators..

Complexity measurement. We speak to multiple complexity models. The data complexity model (big-$\mathcal{O}$ notation) has become the model of choice in the study of conjunctive query processing [4, 58, 37]. The data complexity model assumes that the size of a query, k, is a constant, making data size, n, the parameter of interest[7]. The standard RAM complexity model [5] and combined [61] (big-O notation) consider both k and n as variables. Under data complexity, the lower bound of any join algorithm is $\Omega(n + r)$ [58] (r is the output size) because the algorithm has to read input relations and produce join output.

3.2 Baseline Algorithms

Besides Yannakakis's algorithm (YA) and its improvement (YA$^+$) introduced in Section 1, we further compare TTJ with in-memory hash-join (HJ) and two representative filter methods: Lookahead Information Passing (LIP) and Predicate Transfer (PT). We introduce each of them in order.

HJ evaluates $\mathcal{Q}$ using pipelining on a left-deep plan with in-memory hash-join operators [29]. 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.

LIP [68, 26, 67] 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 [65] 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 $\mathcal{T}_\mathcal{Q}$ 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.

3.3 Additional Definitions

We further define some terminologies used in the paper. We call a relation *internal* if it appears as an internal node [19, 51] in $\mathcal{T}_\mathcal{Q}$. For relations corresponding to non-root internal nodes of $\mathcal{T}_\mathcal{Q}$, we call them *internal*$^\circ$ *relations*. Similarly, a *leaf relation* means the relation appears as a leaf node in $\mathcal{T}_\mathcal{Q}$. The *root relation* is defined accordingly. Let $\mathcal{P}_\mathcal{Q}$ be a left-deep query plan using TTJ. R_i for $i \in [k]$ are relations in $\mathcal{P}_\mathcal{Q}$. The left-most relation is R_k. See Figure 4 (c) in Appendix A. $\bowtie_i$ for $i \in [k]$ are join operators in $\mathcal{P}_\mathcal{Q}$. $\bowtie_1$ is the root operator. $\bowtie_k$ is the table scan operator of R_k. R_{inner} and R_{outer} are right child and left child of $\bowtie_i$, respectively. Depending on context, we adopt 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 $\bowtie_i$, we call it a *join fails at* $\bowtie_i$, a *join failure happens at* $\bowtie_i$, or *join fails at* R_i. Since TTJ determine dangling tuples while doing the join, we consider join failure and detecting a dangling tuple during a join are synonymous. In such case, R_i is called the *detection relation*. $\bowtie_i$ 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 $\mathcal{T}_\mathcal{Q}$. Such R_{inner} is the *guilty relation*. For example, for the join failure happens at $\bowtie_1$ in Figure 1 (b), the detection relation is R and the detection operator is $\bowtie_1$. S is the guilty relation and $\bowtie_3$ is the removal operator.

We introduce extra notation in the paper. Suppose a full ACQ $\mathcal{Q}$ has k relations with each size $O(n)$. The output size of evaluating $\mathcal{Q}$ on a database instance is r. $[R_k, R_{k-1}, \ldots, R_1]$ denotes a query plan $(\ldots((R_k \bowtie R_{k-1}) \bowtie R_{k-2})\ldots) \bowtie R_1$. Let J_u^* denote the join of relations $R_k, R_{k-1}, \ldots, R_u$. Let J_u for $u \in [k]$ denote the join result computed from $\bowtie_u$. Once the correctness of TTJ is proved, $J_u = J_u^*$. $|J_u| = j_u$. R^* is R that is free of dangling tuples w.r.t $\mathcal{Q}$. $t[a] = \pi_a(t)$ for tuple t, attribute a, and projection π. $ja(R, S) = attr(R) \cap attr(S)$. $R(3, 2)$ means tuple $(3, 2) \in R$. $jav(t, R, S) = t[attr(R) \cap attr(S)]$, which is *join-attribute value*. $\mathcal{H}_R$ (or $\mathcal{H}_i$) is the hash table built from R (or associated with $\bowtie_i$). *MatchingTuples* is the list of tuples with the same jav in a hash table. ng is the no-good list, a filter in TTJ scan. $\mathbb{R}$ emphasizes the physical aspects of R, i.e., a bag of tuples R contains. We use standard relational algebra notation, e.g., antijoin $\overline{\ltimes}$ and semijoin $\ltimes$.

4 TreeTracker Join Operators

The pseudo-code presented in Algorithms 4.1 and 4.2 contains the full definition of TTJ. Algorithm 4.1is the primary join algorithm in the context of a left-deep plan. Algorithm 4.2 defines TTJ *scan*. The join operator proper only removes dangling tuples from it's right-hand argument and no changes to a conventional table-scan are needed. However, to attain the complexity results dangling tuples must be removed from the leftmost argument. Thus, TTJ *scan* replaces the scan for the leftmost argument and provides for recording the identity of dangling tuples from the left-most argument and filters them out. We use $\mathcal{P}_\mathcal{Q}$ to denote the left-deep plan using TTJ. We are now ready to work out Example 2 in full detail. We expand

Algorithm 4.1 TTJ Join Operator

Purpose: An iterator returns, one at a time, the join result of R_{outer} and R_{inner}.
Output: A tuple $t \in R_{outer} \ltimes R_{inner}$

```
1  TTJOperator
2    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
3      Initialize r_outer, MatchingTuples to nil
4      R_inner.open()
5      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
6      R_outer.open()
7    Tuple getNext()
8      if MatchingTuples ≠ nil ∧ MatchingTuples ≠ ∅ then
         // If there are more matching tuples left, return the join of r_outer and
            the next matching tuple
9        if (aMatchingTuple ← MatchingTuples.next() ) ≠ nil then
10         return the join of r_outer and aMatchingTuple
         // No matching tuples are left.  Get a new r_outer
11       r_outer ← R_outer.getNext()
12       if r_outer = nil then return nil
13     if r_outer = nil then r_outer ← R_outer.getNext()
14     while r_outer ≠ nil do
         // Find tuples from R_inner joinable with r_outer
15       MatchingTuples ← H.get(jav(r_outer, R_outer, R_inner))
16       if MatchingTuples ≠ nil then
17         aMatchingTuple ← MatchingTuples.next()
18         return the join of r_outer and aMatchingTuple
19       else
           // Join failure identified; start the backjumping to the guilty
              relation, parent of R_inner in T_Q
20         r_outer ← R_outer.deleteDT(R_inner)
21     return nil
22   Tuple deleteDT(Detection Relation R)
23     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
24       Remove aMatchingTuple from MatchingTuples and H
25     else
         // Has not reached the guilty relation for R; backjumping continues
26       MatchingTuples ← nil
27       r_outer ← R_outer.deleteDT(R)
28       if r_outer = nil then return nil
29     return getNext()
```

Algorithm 4.2 TTJ Table Scan Operator for R_k

Purpose: Table scan operator for R_k that returns tuples not in ng.

```
1  TTJScan
2      void open()
3          Initialize ng to an empty set
4      Tuple getNext()
5          while (t ← R_k.next()) ≠ nil do
6              if jav(t, R_k, R_i) ∉ ng for all children R_i of R_k in T_Q then
7                  return t
8          return nil
9      Tuple deleteDT(Detection Relation R)
           // R_k is the guilty relation; t contributes to the tuple that caused the
              join failure at R
10         Insert jav(t, R_k, R) into ng
11         return getNext()
```

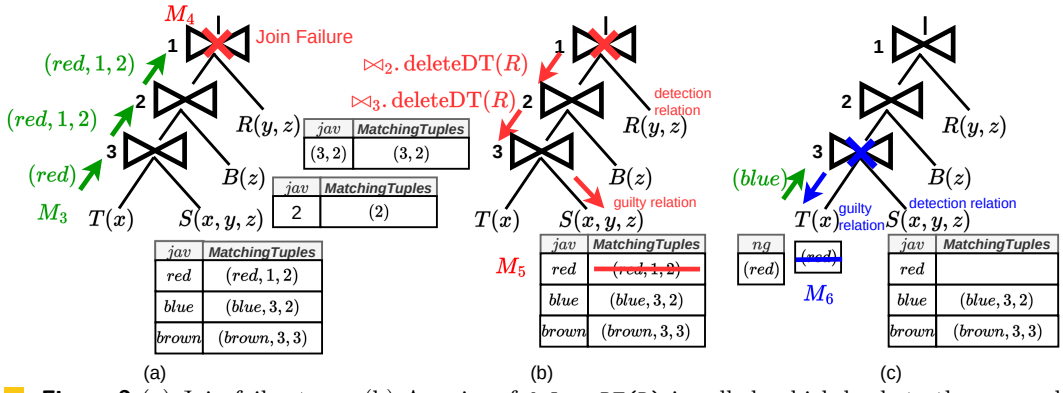

Figure 2 (a) Join fails at $\bowtie_1$. (b) A series of `deleteDT(R)` is called, which leads to the removal of $S(red, 1, 2)$ from hash table $\mathcal{H}_S$. (c) Join further fails at $\bowtie_3$, which puts $T(red)$ to ng.

Figure 1 (b) into Figure 2. By default all line numbers reference Algorithm 4.1 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_3 in Figures 1 and 2). After plan evaluation begins, the recursive `getNext()` calls start with $\bowtie_1$ and end with T's TTJ scan operator (Line 4 Algorithm 4.2), 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 $\bowtie_1$.

On join failure, TTJ needs to reset evaluation flow back to guilty relation as shown in Examples 2 and 4. To do so, we enhance the iterator interface with one more method

`deleteDT()` and reset the evaluation flow using a series of `deleteDT()` calls [3] from the detection operator to the removal operator corresponding to a join failure.

▶ **Example 4** (M_4 and M_5 in Figures 1 and 2)**.** Since $(red, 1, 2)$ cannot join with any tuples from $\mathcal{H}_R$, the goal of TTJ is to reset the evaluation flow back 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 $\mathcal{T}_\mathcal{Q}$ (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 [4] is implemented by removing them from an index. For the left-most argument TTJ scan simply inserts dangling tuples into the *no-good list* (ng). Reads of the left-most argument check for membership in the no-good list and if a tuple is a member the tuple is simply not returned (Example 5).

▶ **Example 5** (M_6 in Figures 1 and 2)**.** 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 4.2 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 4.2 Line 11). The next tuple $T(blue)$ then probes into ng (Algorithm 4.2 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 $\bowtie_3$.

From the above examples, we see that TTJ requires both $\mathcal{P}_\mathcal{Q}$ and $\mathcal{T}_\mathcal{Q}$ to work. The key property that $\mathcal{P}_\mathcal{Q}$ and $\mathcal{T}_\mathcal{Q}$ need to meet is that TTJ can find the guilty relation given a join failure. The following definition and corollary specifies the property.

▶ **Definition 6** (join tree assumption)**.** *Suppose $\mathcal{P}_\mathcal{Q} = [R_k, R_{k-1}, \ldots, R_1]$.* TTJ *assumes $\mathcal{T}_\mathcal{Q}$ satisfies the following property: for a given relation R_i in $\mathcal{P}_\mathcal{Q}$, its parent in $\mathcal{T}_\mathcal{Q}$ is one of the relations $R_k, R_{k-1}, \ldots, R_{i+1}$. The root of $\mathcal{T}_\mathcal{Q}$ is the left-most relation R_k.*

▶ **Corollary 7** (join order view of Definition 6)**.** *Given a $\mathcal{T}_\mathcal{Q}$,* 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 $\mathcal{T}_\mathcal{Q}$, R_i is before R_j in $\mathcal{P}_\mathcal{Q}$, i.e., $\mathcal{P}_\mathcal{Q} = [\ldots, R_i, \ldots, R_j, \ldots]$.*

We use the following lemma to show that Definition 6 is easy to satisfy.

▶ **Lemma 8.** *For any left-deep plan without cross-product for acyclic queries, there exists a $\mathcal{T}_\mathcal{Q}$ satisfies the join tree assumption (Definition 6).*

We defer the proof of Lemma 8 and related examples that illustrate Definition 6 and Corollary 7 to Appendix B.

5 Correctness of TTJ

We prove the correctness of TTJ in this section. The main result in this section is the proof of Theorem 9 which asserts the correctness of TTJ.

[3] We omit argument to `deleteDT()` when reference it generically.
[4] 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 $\mathcal{T}_\mathcal{Q}$.

▶ **Theorem 9 (Correctness of TTJ).** *Evaluating an ACQ of k relations using $\mathcal{P}_\mathcal{Q}$, which consists of $k-1$ instances of Algorithm 4.1 as the join operators and 1 instance of* TTJ *scan (Algorithm 4.2) for the left-most relation R_k, computes the correct query result.*

To prove Theorem 9, we first prove two lemmas that concern identifying join failure.

Let *iter* be an iterator on *MatchingTuples*, i.e., when calling `next()` on *MatchingTuples*, *iter* is advanced and returns the next tuple in *MatchingTuples* if such tuple exists and *nil* otherwise.

▶ **Lemma 10.** *For every value assignment to r_{outer}, MatchingTuples is initialized with tuples from $\mathcal{H}$ and implicitly, iter is reset. Between each pair of value assignments to r_{outer}, MatchingTuples is never initialized and iter is never reset.*

Proof. r_{outer} is assigned in four places: Lines 11, 13, 20, and 27. For Lines 11, 13, and 20, *MatchingTuples* is initialized on Line 15. For Line 27, since *MatchingTuples* is set to *nil* (Line 26), *MatchingTuples* is initialized on Line 15 as well. Since *MatchingTuples* is never initialized with tuples from $\mathcal{H}$ in the rest of Algorithm 4.1, the claim follows. ◀

▶ **Lemma 11.** *A tuple, t, is part of the final join result if and only if it is not marked as dangling during the query evaluation by `deleteDT()`.*

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

We are ready to prove our main theorem Theorem 9.

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

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

If there doesn't exist a relation R in $\mathcal{Q}$ such that $t[attr(R)] \in R$, it is trivial to see that $t \notin J_1$. Suppose $t[attr(R_u)] \in R_u$ for $u = \{k, k-1, \ldots, i+1\}$ but $t[attr(R_u)] \notin R_u$ for $u = \{i, i-1, \ldots, 1\}$. By default join order (Definition 6), R_i must be a child of some relation R_j with $i < j$ such that $t[attr(R_j)] \in R_j$ and $t[attr(R_i)] \notin R_i$. By $\mathcal{T}_\mathcal{Q}$ definition, $attr(R_i) \cap attr(R_j) \neq \emptyset$. The only non-trivial reason that $t[attr(R_i)] \notin R_i$ is because $t[attr(R_i) \cap attr(R_j)] \notin \pi_{attr(R_i) \cap attr(R_j)}(R_i)$. In such case, TTJ will call `deleteDT()` from the join operator connected with R_i and $t[attr(R_j)]$ will be deleted from $\mathcal{H}_{R_j}$ or put onto *ng*. Thus, t is not in J_1. If there is a relation R_u with $k \leq u \leq i+1$ such that $t[attr(R_u)] \notin R_u$, $t \notin J_1$ by the definition of join. The same argument applies to any t whose value is duplicated.

To show $J_1^* \subseteq J_1$, suppose $t \in J_1^*$ but $\notin J_1$. $t[attr(R_1)]$ is part of the join result and with Lemma 11, $t[attr(R_1)]$ is never deleted. Thus, it must be that $t[attr(\mathcal{P}_{\bowtie_2})] \in J_2^*$ but $t \notin J_2$. The same argument applies to every operator in the plan. Eventually, we have $t[attr(R_k)] \in J_k^*$ 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, with Lemma 11, $t[attr(R_k)] \notin ng$ and $\in J_k$.

Next, we show $|J_1| = |J_1^*|$. That is, for a given $t \in J_1^*$, we show the number of tuples t that are in J_1^* equals to the number of tuples t in J_1. By Lemma 11, TTJ will not falsely remove a tuple t that is in J_1^* and if t is a dangling tuple, it is removed by `deleteDT()`. Further, by Lemma 10, each tuple from $\bowtie_u \bowtie R_{u-1}$ is enumerated once. The claim holds. ◀

6 Optimality of TTJ

The runtime analysis of evaluating $\mathcal{P}_\mathcal{Q}$ 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. Clean state permits the existence of more dangling tuples than what is allowed by YA$^+$. Second, we show $\mathcal{P}_\mathcal{Q}$ 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 $\mathcal{O}(n+r)$.

6.1 Clean State

▶ **Definition 12 (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 $\widetilde{\mathbb{R}}_i$:*

1. *$\widetilde{\mathbb{R}}_i = \mathbb{R}_i$ for all the leaf relations R_i of $\mathcal{T}_\mathcal{Q}$;*
2. *$(\mathbb{R}_i \ltimes J^*_{i+1}) \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u = \emptyset$ for internal° relations R_i and their child relations R_u; and*
3. *$\mathbb{R}_k \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u = \emptyset$ for the root of $\mathcal{T}_\mathcal{Q}$, 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 13.** *When the left-deep plan without cross-product for ACQ is in clean state, R_k is fully reduced and free of dangling tuples.*

Proof. Suppose $\mathcal{P}_\mathcal{Q}$ is in clean state. Assume there is a dangling tuple $d \in R_k$. Suppose $\{d\} \ltimes R_{k-1} \ltimes \ldots \ltimes R_j$ but cannot join with R_{j+1} with $j \in \{k-1, \ldots, 2\}$. Given $\mathcal{P}_\mathcal{Q}$ satisfying Definition 6, parent of R_{j+1}, R_i, must be one of the relations joinable with $\{d\}$. Thus, R_i is not in clean state. Contradiction. ◀

▶ **Theorem 14 (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.*

Proof. Proof by induction on the height of $\mathcal{T}_\mathcal{Q}$, h. Base case $h = 0$. Claim trivially holds. Suppose the claim holds for height of $\mathcal{T}_\mathcal{Q} < h$. Let R_h be the root of $\mathcal{T}_\mathcal{Q}$ with height h. Let R_j be a child of R_h. With Lemma 13, no dangling tuples produced when R_h join with R_j. By induction assumption, no dangling tuple produced when further join $R_h \ltimes R_j$ with relations in subtree rooted in R_j. Repeat the same argument for each child of R_h and the result follows. Notice the order of R_js that invoke proof arguments is specified by the order in $\mathcal{P}_\mathcal{Q}$, which satisfies Corollary 7. ◀

Comparison with full reducer and reducing semijoin program. Relations that are free from dangling tuples are in clean state. Thus, relations after $F_\mathcal{Q}$ are in clean state. Relations after $HF_\mathcal{Q}$ are in clean state as well. Leaf relations after $HF_\mathcal{Q}$ satisfy Item 1 (by definition of $HF_\mathcal{Q}$) and the root relation after $HF_\mathcal{Q}$ satisfies Item 3 (by Lemma 13 and Lemma 4 of [12]). For an internal° relation R_i, it satisfies $\mathbb{R}_i \,\overline{\ltimes}\, \widetilde{\mathbb{R}}_u = \emptyset$, which implies the satisfaction of Item 2. However, the state of relations after $HF_\mathcal{Q}$ or $F_\mathcal{Q}$ 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 $F_\mathcal{Q}$ and $HF_\mathcal{Q}$ if such tuples are not joinable with tuples from any child relation of R_i. For example, $S(brown, 3, 3)$ in Example 2. But, those dangling tuples are allowed to present in clean state. We provide one more example in Appendix C.

▶ **Corollary 15.** *The set of dangling tuples removed by* TTJ *is a subset of the set of dangling tuples removed by both* YA *and* YA$^+$.

We also perform empirically measurements on two standard benchmarks to illustrate Corollary 15. The result is in Appendix I.

6.2 Complexity Analysis

▶ **Lemma 16.** *Algorithm 4.2 Line 10 is executed whenever* $\mathbb{R}_k \overline{\ltimes} \mathbb{R}_u \neq \emptyset$ *for child relation* R_u *of* R_k*. Similarly, Line 24 is executed whenever* $\mathbb{R}_i \overline{\ltimes} \mathbb{R}_u \neq \emptyset$ *for internal° relations* R_i *and its child* R_u*.* $\mathbb{R}_u$ *indicates the content of* R_u *can change during* TTJ *execution.*

Proof. We prove the claim on Algorithm 4.2 Line 10; claim on Line 24 can be proved similarly. $t \in R_k$ can be dangling for two reasons. First, t is dangling at the very beginning of the execution, i.e., $\{t\} \overline{\ltimes} R_u = \{t\}$. Then, during the execution with t from $\bowtie_k$, join fails at $\bowtie_u$, and `deleteDT()` is initiated (Line 20). Since R_k is the parent of R_u, Algorithm 4.2 Line 10 is executed. Second, t becomes dangling after all tuples from $R_u \ltimes \{t\}$ are removed. After the last tuple in $R_u \ltimes \{t\}$ is removed by Line 24, *MatchingTuples* becomes empty at $\bowtie_u$. Line 29 is then called. Since $\mathit{MatchingTuples} = \emptyset$ and $R_u \ltimes \{t\} = \emptyset$, Line 15 is executed and returns *nil*. `deleteDT()` is initiated and Algorithm 4.2 Line 10 will be executed. ◀

▶ **Lemma 17.** *When* TTJ *finishes execution,* $\mathcal{P}_\mathcal{Q}$ *is in clean state.*

Proof. *Satisfaction of Item 1.* Suppose R_i is a leaf relation. Since relations that have tuples removed or put into ng are parent of some other relations in $\mathcal{T}_\mathcal{Q}$, condition holds.

Satisfaction of Item 2. Start with internal° relations R_i that are parent of leaf relations R_u. Then, $\mathbb{R}_u = \widetilde{\mathbb{R}}_u$. By Lemma 11 and parent-child relation between R_i and R_u, $(\mathbb{R}_i \ltimes J^*_{i+1}) \overline{\ltimes} \widetilde{\mathbb{R}}_u$ is empty. Thus, $\mathbb{R}_i = \widetilde{\mathbb{R}}_i$ when TTJ finishes execution. Now, let R_i be an internal° relation and R_u be its child, which is also an internal° relation. Start R_u be the parent of leaf relations and apply the same argument from the previous case. $\mathbb{R}_i = \widetilde{\mathbb{R}}_i$. Repeat the same argument all the way till R_u be the grandchild of R_k.

Satisfaction of Item 3. By Lemma 13, equivalently, we show $\mathbb{R}_k - ng = \mathbb{R}^*_k$. First, $\mathbb{R}^*_k \subseteq \mathbb{R}_k - ng$. Suppose $t \notin \mathbb{R}_k - ng$. This means t is one of the tuples removed by Algorithm 4.2 Line 10. With Lemma 16, $t \notin \mathbb{R}^*_k$. Second, $\mathbb{R}^*_k \supseteq \mathbb{R}_k - ng$. Suppose $t \notin \mathbb{R}^*_k$. Then, t has to be a dangling tuple causes a join failure at some relation R. By the proof of Lemma 16, either `deleteDT()` is called directly (R is a child of R_k) or indirectly (R causes all tuples from R_u, a child of R_k, joining with t removed). Thus, $t \notin \mathbb{R}_k - ng$. ◀

▶ **Lemma 18.** TTJ *evaluates* $\mathcal{P}_\mathcal{Q}$ *in* $\mathcal{O}(n + r)$ *once it is in clean state.*

Proof. By Theorem 14, once $\mathcal{P}_\mathcal{Q}$ reaches clean state, no dangling tuple is produced by $\bowtie_u$ for $u \in [k]$. Thus, no more calls on `deleteDT()`. There are k relations and $k - 1$ join operators, `open()` takes $O(kn)$ as each operator is called once and takes $O(n)$ to build $\mathcal{H}$. It takes $O(k)$ `getNext()` calls to compute a tuple in J^*_1. Since each `getNext()` call takes $O(1)$, it takes $O(k)$ to compute one join result and $O(kr)$ for J^*_1. Thus, in total, we have $O(kn + kr) = \mathcal{O}(n + r)$. ◀

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 19 (Data complexity optimality of TTJ).** *Evaluating an ACQ of k relations using $\mathcal{P}_\mathcal{Q}$, which consists of $k-1$ instances of Algorithm 4.1 as the join operators and 1 instance of* TTJ *scan (Algorithm 4.2) 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 17, the execution of a plan is in clean state when TTJ execution finishes. The amount of work that makes $\mathcal{P}_\mathcal{Q}$ 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 $\leq \sum_{i=1}^{k-2}(k-i)\cdot n = O(k^2 n)$.

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

Summing everything together, cleaning cost is $O(k^2 n)$. Since $\mathcal{P}_\mathcal{Q}$ is clean after computing the first join result, with Lemma 18, 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 $\mathcal{P}_\mathcal{Q}$. We defer the details to Appendix D. In Appendix E we show how to use TTJ on plan that is no longer degenerate. We further analyze its performance and formulate a graph mapping problem that can lead to optimality. Lastly, besides the hypertree decomposition approach, TTJ can be easily extended to cyclic queries using the spanning tree approach described in [21], which we detail in Appendix F.

7 Empirical Results

So far, we have shown that by combining two logical operations into one, we maintain the optimality guarantee. To finish our argument, we need to supplement empirical evidence to show our approach gives better overall performance compared to the existing approaches. In this section, we compare the performance of TTJ with the baseline algorithms (Section 3.2) on two standard benchmarks: TPC-H [57], and Star Schema Benchmark (SSB) [46]. We defer

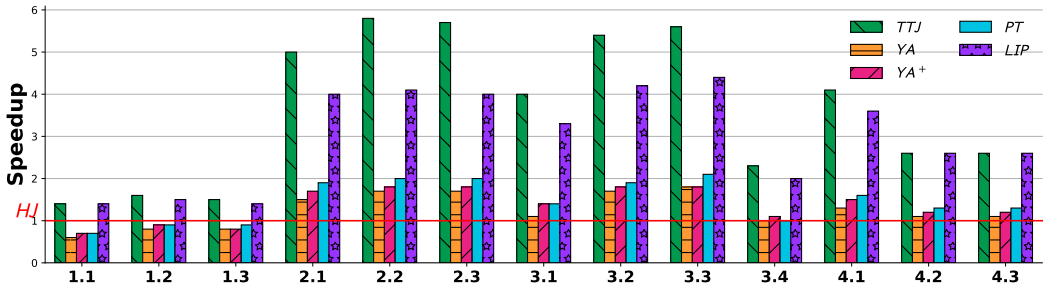

Figure 3 Speedup of TTJ, YA, YA$^+$, PT, and LIP over HJ on all 13 SSB queries

the details of our experimental setup such as algorithm implementation details, workload, environment, and cost models to Appendix G. We present our SSB results and defer TPC-H result to Appendix H.

We present SSB results because star schema queries eliminate the impact of join order and join tree on algorithms' performance; all algorithms share the identical $\mathcal{T}_\mathcal{Q}$ 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 3 illustrates that TTJ has the largest speedup, 3.2× on average, for all SSB queries and LIP comes in second with average of 2.8×. 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 $\mathcal{T}_\mathcal{Q}$. The comparison result between TTJ and LIP supports our argument that the users do not need to trade-off optimality guarantee for empirical performance; they can have both at the same time. Furthermore, in this setup, TTJ is indeed reduced to combining the bottom-up semijoin pass and join pass into one [5], which makes TTJ equivalent to YA$^+$. However, TTJ outperforms YA$^+$, which shows that TTJ is more empirical efficient than YA$^+$ despite the equivalent formal runtime guarantee. Compared with LIP and YA$^+$, YA and PT perform an additional top-down pass of $\mathcal{T}_\mathcal{Q}$. PT has extra pass compared to YA$^+$ but still outperforms YA$^+$ in most cases; such results reflect using Bloom filter is much more cost-effective than using semijoin. Additional results on the number of dangling tuples removed by each algorithm is in Appendix I.

8 Limitations and Future Work

In this paper, we use TTJ to argue that two separate logical operations: semijoin and join, is the fundamental reason that users have had to pick between theoretical guarantee and good empircal performance. Theoretical gaps remain when consider TTJ with additional requirements, which we discuss next. First, the combined complexity of TTJ can be improved because it has an additional $\log k$ term compared with the complexity of YA. Second, optimality of TTJ under bushy plan is contingent upon the resolution of a graph mapping problem that maps $\mathcal{T}_\mathcal{Q}$ into query plan.

References

1 Java Microbenchmark Harness (JMH). URL: `https://github.com/openjdk/jmh`.

[5] For other ACQs, reduction does not hold.

544 **2** Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and
545 Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans.*
546 *Database Syst.*, 42(4), October 2017. URL: https://doi-org.ezproxy.lib.utexas.edu/10.
547 1145/3129246, doi:10.1145/3129246.
548 **3** Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*, volume 8.
549 Addison-Wesley Reading, 1995.
550 **4** Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What Do Shannon-Type Inequal-
551 ities, Submodular Width, and Disjunctive Datalog Have to Do with One Another? In
552 *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database*
553 *Systems*, PODS '17, page 429–444, New York, NY, USA, 2017. Association for Computing
554 Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3034786.3056105,
555 doi:10.1145/3034786.3056105.
556 **5** Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Design and Analysis of Computer*
557 *Algorithms*. Addison-Wesley, 1974.
558 **6** Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic,
559 Todd L Veldhuizen, and Geoffrey Washburn. Design and Implementation of the LogicBlox
560 System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management*
561 *of Data*, pages 1371–1382, 2015.
562 **7** Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database*
563 *Theory*. Open source at https://github.com/pdm-book/community, 2022.
564 **8** Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic Sets and
565 Other Strange Ways to Implement Logic Programs (Extended Abstract). In *Proceedings of*
566 *the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86,
567 page 1–15, New York, NY, USA, 1985. Association for Computing Machinery. URL: https:
568 //doi-org.ezproxy.lib.utexas.edu/10.1145/6012.15399, doi:10.1145/6012.15399.
569 **9** Roberto J. Bayardo Jr and Daniel P. Miranker. An Optimal Backtrack Algorithm for Tree-
570 Structured Constraint Satisfaction Problems. *Artificial Intelligence*, 71(1):159–181, 1994.
571 **10** Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability
572 of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, July 1983. URL: https://doi-org.
573 ezproxy.lib.utexas.edu/10.1145/2402.322389, doi:10.1145/2402.322389.
574 **11** Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire.
575 Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogen-
576 eous Data Sources. In *Proceedings of the 2018 International Conference on Management of*
577 *Data*, SIGMOD '18, page 221–230, New York, NY, USA, 2018. Association for Computing
578 Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3183713.3190662,
579 doi:10.1145/3183713.3190662.
580 **12** Philip A. Bernstein and Dah-Ming W. Chiu. Using Semi-Joins to Solve Relational Queries. *J.*
581 *ACM*, 28(1):25–40, January 1981. doi:10.1145/322234.322238.
582 **13** Philip A Bernstein and Nathan Goodman. Power of Natural Semijoins. *SIAM Journal on*
583 *Computing*, 10(4):751–771, 1981.
584 **14** Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun.*
585 *ACM*, 13(7):422–426, jul 1970. doi:10.1145/362686.362692.
586 **15** Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries
587 in Relational Data Bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory*
588 *of Computing*, STOC '77, page 77–90, New York, NY, USA, 1977. Association for Comput-
589 ing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/800105.803397,
590 doi:10.1145/800105.803397.
591 **16** Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings*
592 *of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database*
593 *systems*, pages 34–43, 1998.
594 **17** Ming-Syan Chen and Philip S. Yu. Using Join Operations as Reducers in Distributed Query
595 Processing. In *Proceedings of the Second International Symposium on Databases in Parallel*

and Distributed Systems, DPDS '90, page 116–123, New York, NY, USA, 1990. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/319057.319074, doi:10.1145/319057.319074.

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

19 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 3rd edition, 2009.

20 Rina Dechter. *Constraint Processing*. Morgan Kaufmann, USA, 2003.

21 Kyle Deeds, Dan Suciu, Magda Balazinska, and Walter Cai. Degree Sequence Bound For Join Cardinality Estimation. *CoRR*, abs/2201.04166, 2022. URL: https://arxiv.org/abs/2201.04166, arXiv:2201.04166.

22 Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. Bitvector-Aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2011–2026, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3318464.3389769, doi: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. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 418–431, New York, NY, USA, 2021. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3448016.3457270, doi:10.1145/3448016.3457270.

24 Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. *ACM J. Exp. Algorithmics*, 26, jul 2021. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3440015, doi:10.1145/3440015.

25 Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.*, 13(12):1891–1904, July 2020. URL: https://doi-org.ezproxy.lib.utexas.edu/10.14778/3407790.3407797, doi:10.14778/3407790.3407797.

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

27 Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2nd edition, 2008.

28 Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, page 57–74, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2902251.2902309, doi:10.1145/2902251.2902309.

29 Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/152610.152611, doi:10.1145/152610.152611.

30 Thomas Mueller Graf and Daniel Lemire. Binary Fuse Filters: Fast and Smaller Than Xor Filters. *ACM J. Exp. Algorithmics*, 27, mar 2022. doi:10.1145/3510449.

31 Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1259–1274, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3035918.3064027.

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

33 Guodong Jin and Semih Salihoglu. Making RDBMSs Efficient on GraphWorkloads Through Predefined Joins. *PVLDB 15*, 2022.

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

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

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

37 Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.*, 41(4), November 2016. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2967101, doi:10.1145/2967101.

38 Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-Query Containment and Constraint Satisfaction. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, page 205–213, New York, NY, USA, 1998. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/275487.275511, doi:10.1145/275487.275511.

39 Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015. URL: https://doi-org.ezproxy.lib.utexas.edu/10.14778/2850583.2850594, doi:10.14778/2850583.2850594.

40 Zhe Li and Kenneth A. Ross. PERF Join: An Alternative to Two-Way Semijoin and Bloomjoin. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, CIKM '95, page 137–144, New York, NY, USA, 1995. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/221270.221360, doi:10.1145/221270.221360.

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

42 Inderpal Singh Mumick and Hamid Pirahesh. Implementation of Magic-Sets in a Relational Database System. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, page 103–114, New York, NY, USA, 1994. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/191839.191860, doi:10.1145/191839.191860.

43 Yoon-Min Nam Nam, Donghyoung Han Han, and Min-Soo Kim Kim. 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*, SIGMOD '20, page 2055–2070, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3318464.3380565, doi:10.1145/3318464.3380565.

44 Thomas Neumann and Gerhard Weikum. Scalable Join Processing on Very Large RDF Graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 627–640, New York, NY, USA, 2009. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/1559845.1559911, doi:10.1145/1559845.1559911.

45 Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. Beyond Worst-Case Analysis for Joins with Minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, page 234–245, New York, NY, USA,

2014. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2594538.2594547, doi:10.1145/2594538.2594547.

46 Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, pages 237–252, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

47 Felix Putze, Peter Sanders, and Johannes Singler. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM J. Exp. Algorithmics*, 14, jan 2010. doi:10.1145/1498698.1594230.

48 Wilson Qin and Stratos Idreos. Adaptive Data Skipping in Main-Memory Systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2255–2256, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2882903.2914836, doi:10.1145/2882903.2914836.

49 Mark Raasveldt and Hannes Mühleisen. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3299869.3320212.

50 Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2nd edition, 2000.

51 Kenneth Rosen. *Discrete Mathematics and Its Applications*. McGraw Hill, 7th edition, 2011.

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

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

54 Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. Materialization Strategies in the Vertica Analytic Database: Lessons Learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1196–1207. IEEE, 2013.

55 Abraham Silberschatz, Henry F. Korth, and Shashank Sudarshan. *Database System Concepts*. McGraw-Hill New York, 7th edition, 2019.

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

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

58 Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal Join Algorithms Meet Top-k. page 2659–2665, 2020. doi:10.1145/3318464.3383132.

59 Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems Vol. 2: The New Technologies*. Computer Science Press, USA, first edition, 1989.

60 Patrick Valduriez and Georges Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Trans. Database Syst.*, 9(1):133–161, mar 1984. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/348.318590, doi:10.1145/348.318590.

61 Moshe Y. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 137–146,

New York, NY, USA, 1982. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/800070.802186, doi:10.1145/800070.802186.

62 Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. Change Propagation Without Joins. *Proc. VLDB Endow.*, 16(5):1046–1058, jan 2023. doi:10.14778/3579075.3579080.

63 Yisu Remy Wang, Max Willsey, and Dan Suciu. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data*, 1(2), jun 2023. doi:10.1145/3589295.

64 Sungheun Wi, Wook-Shin Han, Chuho Chang, and Kihong Kim. Towards Multi-Way Join Aware Optimizer in SAP HANA. *Proc. VLDB Endow.*, 13(12):3019–3031, August 2020. doi:10.14778/3415478.3415531.

65 Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. *Conference on Innovative Data Systems Research (CIDR)*, 2024. arXiv:2307.15255.

66 Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, volume 81, pages 82–94, 1981.

67 Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proc. VLDB Endow.*, 16(11):2962–2975, jul 2023. doi:10.14778/3611479.3611501.

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

A Illustration of Notation

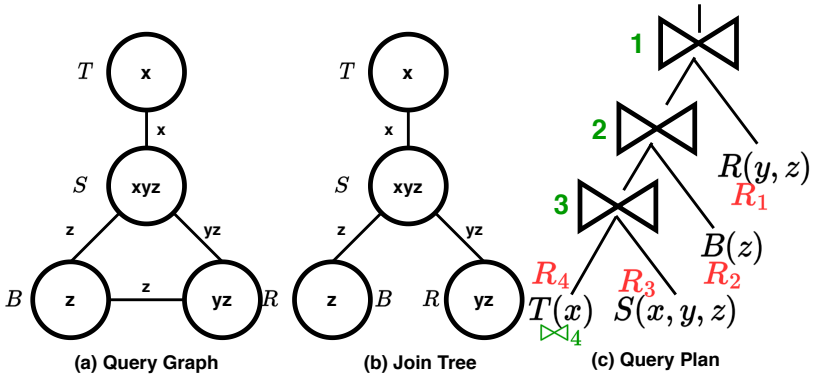

Figure 4 (a) query graph, (b) join tree , and (c) query plan of $\mathcal{Q}$ of four relations in (c). $R_1, \ldots, R_4$ show the relation numbering and $\bowtie_1, \bowtie_2, \bowtie_3, \bowtie_4$ denote the join operator numbering. $\bowtie_4$ represents the table scan operator associated with the left-most relation R_4, which is T in this example.

B Understanding TTJ Requirements on $\mathcal{P}_\mathcal{Q}$ and $\mathcal{T}_\mathcal{Q}$

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 $\mathcal{T}_\mathcal{Q}$ to find the parent of the detection relation, i.e., the guilty relation, for a join failure. Thus, if either the plan or the $\mathcal{T}_\mathcal{Q}$ is missing, we need to construct it from the other one. A constraint exists for such construction to ensure TTJ can function correctly. 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 $\mathcal{T}_\mathcal{Q}$ is missing, we need to construct a $\mathcal{T}_\mathcal{Q}$ 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 $\mathcal{T}_\mathcal{Q}$ or a plan given the other input.

Given $\mathcal{P}_\mathcal{Q}$, Definition 6 defines the aforementioned constraint on the $\mathcal{T}_\mathcal{Q}$. calling

▶ **Example 20.** Consider $\mathcal{P}_\mathcal{Q}$ in Figure 4 (c), B is labeled as R_2. TTJ expects that B's parent in $\mathcal{T}_\mathcal{Q}$ has to be either R_3 or R_4. As shown in Figure 4 (b), B's parent is S, which corresponds to R_3. Thus, $\mathcal{T}_\mathcal{Q}$ in (b) satisfies the assumption.

Lemma 8 states that we can easily construct a required $\mathcal{T}_\mathcal{Q}$ from any left-deep query plan that does not have cross-product. The key idea is as follows: We construct $\mathcal{T}_\mathcal{Q}$ following the order of relations in $\mathcal{P}_\mathcal{Q}$ from left to right. Suppose $R_k, \ldots, R_{j+1}$ are already added to $\mathcal{T}_\mathcal{Q}$. For R_j, we want to find a relation R_i that is already in $\mathcal{T}_\mathcal{Q}$ 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 $\mathcal{T}_\mathcal{Q}$ through an edge (R_i, R_j).

▶ **Example 21.** Suppose $\mathcal{P}_\mathcal{Q} = [R_3(x, y), R_2(x, y, z), R_1(y, z)]$. The left-most relation $R_3(x, y)$ has to be the root of $\mathcal{T}_\mathcal{Q}$. For the next relation $R_2(x, y, z)$, since only R_3 is in $\mathcal{T}_\mathcal{Q}$ 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 $\mathcal{T}_\mathcal{Q}$ 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 $\mathcal{T}_\mathcal{Q}$ is $R_3 \to R_2 \to R_1$.

▶ **Example 22.** Consider a cyclic query, $\mathcal{P}_\mathcal{Q} = [R_3(a, b), R_2(b, c), R_1(c, a)]$, the classic triangle query. Let us try to construct $\mathcal{T}_\mathcal{Q}$. $R_3(a, b)$ is the root. $R_2(b, c)$ connects R_3.

$attr(R_3) \cup attr(R_2) = \{a, b, c\}$. But, $attr(R_1) \cap \{a, b, c\} \not\subseteq attr(R_2)$ and $attr(R_1) \cap \{a, b, c\} \not\subseteq attr(R_3)$. R_1 cannot be placed in $\mathcal{T}_\mathcal{Q}$ to satisfy the connectedness property while keeping $\mathcal{T}_\mathcal{Q}$ being a tree.

▶ **Example 23.** $\mathcal{P}_\mathcal{Q} = [T(x), R(y, z), B(z), S(x, y, z)]$ contains a cross-product due to $T(x), R(y, z)$. We cannot construct $\mathcal{T}_\mathcal{Q}$ because $\mathcal{T}_\mathcal{Q}$ is a subgraph of the query graph and the query graph does not contain (T, R) edge.

Proof. For a left-deep plan without cross-product for an acyclic CQ $[S_k, S_{k-1}, \ldots, S_1]$, our proof proceeds by showing the plan permits a rooted join tree that satisfies Definition 6. That is, S_k is R_k, the root of some $\mathcal{T}_\mathcal{Q}$, and for any relation S_i, its parent in $\mathcal{T}_\mathcal{Q}$ is S_j with $j \in \{k, k-1, \ldots, i+1\}$. For a relation S_i, let attribute set $as(S_i)$ denote the set of attributes appear before it in the plan, i.e., $as(S_i) = attr(S_k) \cup \cdots \cup attr(S_{i+1})$. The plan has the property that $attr(S_i) \cap as(S_i) \neq \emptyset$. We want to show there is some relation S_j with $j \in \{k, k-1, \ldots, i+1\}$ such that $attr(S_j) \supseteq (attr(S_i) \cap as(S_i))$. If the statement is true, we can construct $\mathcal{T}_\mathcal{Q}$ by adding edge (S_j, S_i). To prove the statement, for a relation S_u, suppose relations before S_u already form a join tree, i.e., we are about to attach S_u to the tree. Suppose the statement is not true and there are two more relations S_i, S_j $(i > j > u)$ in the plan such that $attr(S_u) \cap as(S_u) = (attr(S_i) \cup attr(S_j))$ and $attr(S_u) \cap as(S_u) \not\subseteq attr(S_i)$ (correspondingly for $attr(S_j)$ as well). S_i and S_j are connected via a path. To satisfy join tree requirement, one must add two edges (S_i, S_u) and (S_j, S_u), which form a cycle. ◀

Definition 6 can be interpreted as a join order assumption, which defines the constraint on the plan (Corollary 7). Construction of $\mathcal{P}_\mathcal{Q}$ is straightforward: performing a top-down pass (not necessarily from left to right) of $\mathcal{T}_\mathcal{Q}$.

▶ **Example 24.** For $\mathcal{T}_\mathcal{Q}$ in Figure 4 (b) with T as the root, both $\mathcal{P}^1_\mathcal{Q} = [T, S, B, R]$ and $\mathcal{P}^2_\mathcal{Q} = [T, S, R, B]$ are valid plans for TTJ.

C An Additional Example on

▶ **Example 25.** Consider a $\mathcal{T}_\mathcal{Q}$ $R_3(x) \to R_2(x, y) \to R_1(y)$ 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 $R_2(4, 6)$. $HF_\mathcal{Q}$ removes two tuples $R_2(4, 6)$ and $R_2(3, 5)$. $F_\mathcal{Q}$ removes three tuples: $R_2(4, 6)$, $R_2(3, 5)$, and $R_2(3, 7)$.

D Improving TTJ Combined Complexity

Theorem 19 gives $O(k^2n+kr)$ combined complexity. We can further improve it to $O(nk \log k + kr)$ by constraining the join order (Corollary 7). In particular, to decide join order, one pre-order traverses $\mathcal{T}_\mathcal{Q}$ and when multiple subtrees exist for a given relation in $\mathcal{T}_\mathcal{Q}$, one breaks ties by visting the largest subtree of any relation last [9]. Figure 5 shows an example.

▶ **Theorem 26 (Improving combined complexity of TTJ).** *Combined complexity of* TTJ *can be improved to* $O(nk \log k + kr)$ *(*log *is base 2) if one performs pre-order traversal over* $\mathcal{T}_\mathcal{Q}$ *and break ties by visiting the largest subtree of any relation last.*

Proof. The new order strategy only changes the total number of `deleteDT()` calls (Line 27) in Theorem 19 proof. For a given $\mathcal{T}_\mathcal{Q}$, let b_i be the backjumping distance where the join failure relation is R_i. Note that b_i is exactly the same as the number of `deleteDT()` calls generated (Line 27) when join fails at $\bowtie_i$. $b_i \leq k - i$ for the default join order. Let d_i denote

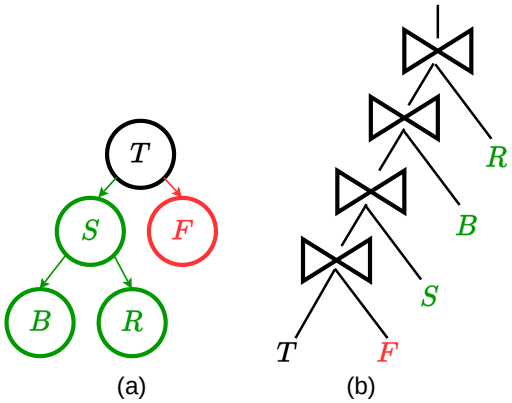

Figure 5 Given $\mathcal{T}_\mathcal{Q}$ in (a), one decides join order by pre-order traversing over $\mathcal{T}_\mathcal{Q}$ and breaking ties via visiting the largest subtree of any relation last. The resulting order (b) satisfies Corollary 7.

the number of descendents of an internal relation R_i and m_i denotes the number of relations in the largest subtree rooted at one of i's children, e.g., in Figure 5, $d_T = 4$ and $m_T = 3$. Since the new order satisfies Definition 6, when join fails at R_i, only descendents of R_j (the parent of R_i) could exist between R_i and R_j in the order. The largest number of `deleteDT()` generated when join fails at the root relation of the largest subtree of a relation. Thus, $b_i \leq d_i - m_i + 1$.

Next, we prove $\sum_{i=1}^{k-1} b_i \leq k \log k$. Proof by induction on the size of $\mathcal{T}_\mathcal{Q}$. Base case $k = 1$, the claim holds. Assuming the claim holds for $k-1$. Suppose there are s subtrees of R_k and each with size $k_1, \ldots, k_s$. Let k_m denote the largest subtree. Then $b_r \leq (k-1) - k_m + 1 = k - k_m$. Thus, $\sum_{i=1}^{k-1} b_i \leq \sum_{i=1}^{s} k_i \log k_i + (k - k_m) \leq k \log k_m + (k - k_m) \leq k \log k$ (the last inequality follows Lemma A.1 in [9]). Then, the total number of `deleteDT()` calls on Line 27 is $\leq \sum_{i=1}^{k-1} b_i n = O(nk \log k)$. ◀

E Bushy Plan

A common approach to evaluate a *bushy plan*, query plan that is no longer degenerate, is to decompose it into a sequence of left-deep subplans: right child of every join operator forms a left-deep subplan and is evaluated first before proceeding with the join [63, 55]. In particular, for in-memory hash-join, build side is a blocking operation, i.e., hash tables can be constructed not just from base relations but also from intermediate results computed from subplans, which are buffered inside the memory [55, 32]. TTJ works with bushy plan exactly as above. The only issue to address is to transform $\mathcal{T}_\mathcal{Q}$ into a bushy plan satisfying Corollary 7 [6] so that when join fails at R, `deleteDT()` can find its parent. We use Algorithm E.1 to control the construction of a bushy plan for TTJ. Such algorithm can be easily adapted into a "reverse-engineer" procedure where one can construct a $\mathcal{T}_\mathcal{Q}$ from the given bushy plan: we construct a join tree for each left-deep subplan using Lemma 8 and concatenate all the trees to form the final join tree.

Fragment group FG is a set of nodes in $\mathcal{T}_\mathcal{Q}$ constituting a subplan. We use FG and subplan interchangeably. Any node from $\mathcal{T}_\mathcal{Q}$ only belongs to one group. The key idea to form a bushy plan is that we create a TTJ-compatible subplan for each group and connect them altogether using TTJ join operators again. Fragment groups are formed with the property

[6] We use join order view of Definition 6.

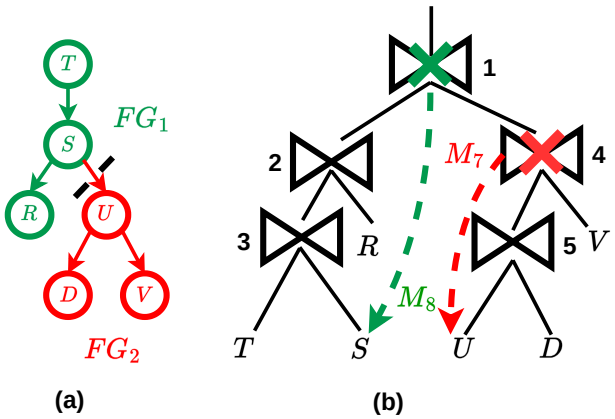

Figure 6 Given $\mathcal{T}_\mathcal{Q}$ in (a), there are two fragment groups FG_1 and FG_2. (b) is a bushy plan constructed from the two fragment groups. Join failures can be categorized into two cases: within FG (M_{13}) and across FGs (M_{14}).

Algorithm E.1 Construct bushy plan for TTJ

Input: $\mathcal{T}_\mathcal{Q}$
Output: A bushy plan that can be evaluated by TTJ

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

that parent node belongs to the same or lower-numbered group than its child node(s) in $\mathcal{T}_\mathcal{Q}$. Line 2 checks sibling node to avoid cross-product when join two subplans. The resulting plan satisfies Corollary 7 and can be evaluated by TTJ directly.

▶ **Example 27.** Consider $\mathcal{Q}$ represented in Figure 6 (a). There are two fragment groups $FG_1 = \{T, S, R\}$ and $FG_2 = \{U, D, V\}$. The whole plan is being evaluated by TTJ operators: every join operator is TTJ join operator; T and U are TTJ table scan operators and the rest are normal table scan operators. `deleteDT()` happens for two cases. First, `deleteDT()` happens inside the subplan. For example, join fails at $\bowtie_4$ (M_7). Since U is the parent of V, a tuple from U is added to ng. Second, `deleteDT()` happens at the join operator connecting two subplans. For example, join fails at $\bowtie_1$ (M_8). In this case, `deleteDT()` sends the reference to the root of FG_2, U, downward. The rest of the evaluation is the same as the left-deep plan case in the previous sections.

▶ **Lemma 28.** *The bushy plan constructed from Algorithm E.1 satisfies Corollary 7.*

Proof. Let S be a node and R, U be its children. There are three possible cases. First, if R and U are all within the same FG as S, by Line 3, the claim holds. Second, if one of

its children is in a different FG, say, U. Since S is in the FG with smaller numbering, by Line 4, S is to the left of U. S is to the left of R because they are in the same FG. Third, if all of its children are in different FGs, by a similar argument as the previous case, the claim holds. ◀

▶ **Theorem 29 (Correctness).** TTJ *evaluates $\mathcal{Q}$ correctly under the bushy plan constructed from Algorithm E.1.*

Proof. By Lemma 28 and Theorem 9, TTJ evaluates relations associated with each FG correctly. We only need to focus on TTJ operators that join two different FGs and show it generates the correct join result. W.l.o.g., the two FGs are denoted FG_1 and FG_2. We want to show $FG_1 \bowtie FG_2$ is correct. If all tuples from FG_1 are joinable with some tuple from FG_2 join result, the claim holds. Suppose $t \in FG_1$ cannot join with any tuple from FG_2 and thus dangling. By Lemma 28, the parent of the node where join fails must be in FG_1. Applying the same arguments in Theorem 9, the claim holds. ◀

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

▶ **Theorem 30 (Data complexity of TTJ on bushy plan).** *Suppose there are m FGs and each has result size $r_1, \ldots, r_m$, respectively.* TTJ *runs $\mathcal{O}(n + \max(r_1, \ldots, r_m))$.*

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

Effectively, Algorithm E.1 maps $\mathcal{T}_\mathcal{Q}$ into a bushy plan, another tree. Together with Theorem 30, we can see that if $r_i \leq r$ for $i \in [m]$. TTJ can provide optimality guarantee under bushy plan. Thus, it is an open question whether we can construct a bushy plan such that the output size of each fragment is bounded by the final output size r. If such algorithm exists, TTJ is optimal under bushy plan as well.

F Handle Cyclic CQ

We show a simple approach to allow TTJ to work with cyclic CQs as well. The key challenge for cyclic CQs is that the query does not permit $\mathcal{T}_\mathcal{Q}$, i.e., a graph that has to contain cycles to satisfy the connectedness property. Let $G_\mathcal{Q}$ denote the graph that contain cycles but satisfies the connectedness property. As an example, (a) in Figure 7 shows $G_\mathcal{Q}$ for a query joining $T(a, b)$, $S(b, c)$, $B(a, c)$, and $R(b)$. The query is a simple extension to the classic triangle query $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$, which makes the query being cyclic. Our solution to the challenge is simple: conceptually, we remove edges from $G_\mathcal{Q}$ to obtain $\mathcal{T}_\mathcal{Q}$ by renaming necessary attributes. Then, we introduce a select operator at the root of the plan for the query to filter out redundant tuples so that the final result satisfies the original query semantics. The idea is the same as the spanning tree approach described in [21].

▶ **Example 31.** Consider the cyclic query shown in Figure 7. Red color indicates the new operations introduced to handle cyclic CQs.

In the example, we remove edge (T, B). Since $attr(B) \cap attr(T) = \{a\}$, we rename attribute a in one of these two relations. In this case, we rename a in B to d. Thus, in the query plan, we introduce $\rho_{B(c,d)}B$ right above B. The resulting query joining $T(a, b)$, $S(b, c)$,

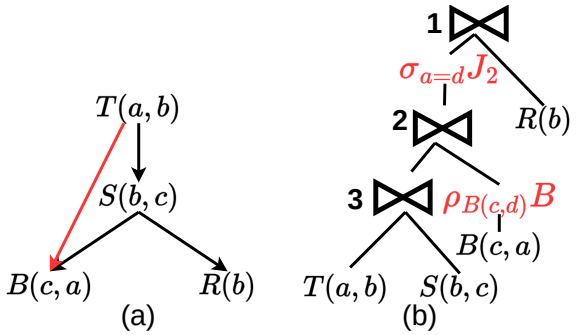

Figure 7 (a) $G_{\mathcal{Q}}$ (b) query plan of $T(a,b) \bowtie S(b,c) \bowtie B(a,c) \bowtie R(b)$.

$B(c,d)$, and $R(b)$ is acyclic and can be evaluated using TTJ. Since cycle in $G_{\mathcal{Q}}$ contains T, S, and B, we add $\sigma_{a=d} J_2$ right above $\bowtie_2$ to filter out those tuples that their a attribute values and d attribute values are different, i.e., adding the removed edge back.

Clearly the simple solution computes the correct join result: the correct join result is subset of the join result compute from $\bowtie_1$. To find $G_{\mathcal{Q}}$ for a given join order, one can build $\mathcal{T}_{\mathcal{Q}}$ and introduce necessary extra edges (and thus, form cycles) to satisfy the connectedness property. Those extra edges will be temporarily removed as illustrated in the example above.

Essentially, the approach uses an acyclic query to contain a given cyclic query (in query containment sense [15]) to compute a superset of the cyclic query result set and removes redundant tuples with selections. Thus, the runtime performance of this approach ties to the runtime performance of evaluating the acyclic query, which is $\mathcal{O}(n + r')$ where r' is the output size of the acyclic query.

G Experiment Setup

G.1 Algorithms and Implementation

We compare TTJ with the baseline algorithms (Section 3.2) 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 [53, 11]. The engine optimizes each algorithm using the same DP procedure [27] with an algorithm-specific cost model (Appendix G.4). The engine connects two data sources: PostgreSQL 13, which provides the estimation to the terms in the cost models, and DuckDB [49], which serves as the storage manager. All data are stored on disk.

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 baseline algorithms that are not described in Section 3.2. To implement YA, we introduce a k-ary physical operator *full reducer operator* that executes $F_{\mathcal{Q}}$. 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 $\mathcal{T}_{\mathcal{Q}}$. Redundant edges may lead to additional unnecessary passes of Bloom filters that may negatively impact PT performance [7]. Thus, we

[7] We conducted an empirical study by comparing PT on the predicate transfer graph with the same PT

show the results of PT on $\mathcal{T}_\mathcal{Q}$. We use the blocked Bloom filter [47] implementation from [30].

G.2 Workload and Environment

Workload. We use three workloads: Join Ordering Benchmark (JOB) [39], TPC-H [57] (scale factor = 1), and Star Schema Benchmark (SSB) [46] (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.

G.3 TTJ Cost Model

Given $\mathcal{P}_\mathcal{Q}$ is a transformation of $\mathcal{T}_\mathcal{Q}$, costing TTJ is the same as costing evaluation of $\mathcal{T}_\mathcal{Q}$ under TTJ. One simple but effective cost model is the sum of the sizes of intermediate results [39, 22, 18, 56], which comprises two components: the dangling tuples produced and the size of intermediate results that are part of final join result. The former corresponds to the cleaning cost in the optimality proof, which can be estimated based on the clean state. Using clean state, the latter can be estimated easily as well. Since TTJ reduces internal° relation sizes, like [22], TTJ cost includes the size of inner relations that are in clean state as well.

▶ **Theorem 32.** *The cost of* TTJ*, i.e., the cost of* $\mathcal{T}_\mathcal{Q}$ *when evaluated by* TTJ*, is*

$$\sum_{i=1}^{m}\sum_{t=0}^{|\mathcal{A}^i|-1} b^{R_i}_{R_u^{t+1}}|(\mathbb{R}_i^{[t]} \ltimes J_{i+1}^*) \overline{\ltimes} \widetilde{\mathbb{R}}_u^{t+1})| \tag{6}$$

$$+\sum_{i=1}^{s}\sum_{t=0}^{|\mathcal{B}^i|-1} b^{R_i}_{R_u^{t+1}}|(\mathbb{R}_i^{[t]} \ltimes J_{i+1}^*) \overline{\ltimes} \widetilde{\mathbb{R}}_u^{t+1})| \tag{7}$$

$$+\sum_{t=0}^{|\mathcal{C}|-1} b^{R_k}_{R_u^{t+1}}|\delta(\pi_{ja(R_k,R_u^t)}(\mathbb{R}_k^{[t]} \overline{\ltimes} \widetilde{\mathbb{R}}_u^{t+1}))| \tag{8}$$

$$+\sum_{j=k}^{1}|f(S_j)| \tag{9}$$

$$+\sum_{i=2}^{k}|\widetilde{\mathbb{R}_i}| \tag{10}$$

Equations (6)–(8) give the number of dangling tuples. Equation (9) gives the size of intermediate results (including the size of R_k^*) that are part of final join result. Equation (10) is the summation of size of internal° and leaf relations that are in clean state.

on $\mathcal{T}_\mathcal{Q}$ to verify our conclusion. Result shows PT on $\mathcal{T}_\mathcal{Q}$ outperforms PT on the predicate transfer graph by 1× (Appendix G.5).

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

$$\mathbb{R}_i^{[t]} = \begin{cases} \mathbb{R}_i & \text{if } t = 0 \\ \mathbb{R}_i^{[t-1]} - ((\mathbb{R}_i^{[t-1]} \ltimes J_{i+1}^*) \overline{\ltimes} \widetilde{\mathbb{R}}_u^t) & \text{otherwise} \end{cases} \tag{11}$$

Suppose the join order (w.r.t. $\mathcal{T}_\mathcal{Q}$) determined either syntactically from $\mathcal{T}_\mathcal{Q}$ or from the DP algorithm is $[S_k, \ldots, S_1]$ where $S_j = R_i$ for some $i \in [k]$. $f(S_j)$ in Equation (12) computes the size of intermediate results that are part of the final join result. Given Definition 6, the first relation to apply f is always R_k, root of $\mathcal{T}_\mathcal{Q}$, and its content, per Lemma 13, is $\mathbb{R}_k^*$.

$$f(S_j) = \begin{cases} \mathbb{R}_k^* & \text{if } j = k \\ \widetilde{\mathbb{S}}_j \bowtie f(S_{j+1}) & \text{otherwise} \end{cases} \tag{12}$$

▶ **Example 33.** Using Example 2, we illustrate how to compute Equations (6)–(8), the most complex terms in the cost equation. Assume $\mathcal{P}_\mathcal{Q} = [T, S, B, R]$. Start with Equation (6). $\mathcal{A} = \{B, R\}$. Since both B and R have the same parent, S, $m = 1$. Thus, the cost is $1 \cdot |(S \ltimes T) \overline{\ltimes} B| + 2 \cdot |(S^{[1]} \ltimes T) \overline{\ltimes} R|$. In particular, $|(S \ltimes T) \overline{\ltimes} B| = 0$. Therefore, $S^{[1]} = S$. $|(S^{[1]} \ltimes T) \overline{\ltimes} R| = 1$ due to $S(red, 1, 2)$. Thus, $\widetilde{S} = \{(red, 3, 2)\}$. Since $\mathcal{B} = \emptyset$, we do not need to compute Equation (7). To compute Equation (8), due to $\mathcal{C} = \{S\}$, we have $1 \cdot |\delta(\pi_x(T \overline{\ltimes} \widetilde{S}))| = 1$. Thus, the number of dangling tuples produced by TTJ is $0 + 2 + 1 = 3$.

G.4 Baseline Algorithms' Cost Models

The cost model of HJ is the summation of intermediate results [39, 27]. Query plan and $\mathcal{T}_\mathcal{Q}$ are fixed for all compared algorithms on star schema queries. Thus, we do not cost LIP. PT shares the same $\mathcal{T}_\mathcal{Q}$ as YA. We detail the cost model of YA below.

The central idea of costing YA is exactly identical to how we cost TTJ in Appendix G.3. We first deduce the state of relations after $F_\mathcal{Q}$ called *full reducer state* (Definition 34), which is similar to clean state (Definition 12). Then, we compute the number of intermediate results produced by $F_\mathcal{Q}$ (Equations (13)–(15)), the size of the intermediate results that are part of the final join result (Equation (16)), and the size of the relations that are in full reducer state (Equation (17)) in YA cost equation (Theorem 35).

▶ **Definition 34 (full reducer state).** *Query plan using $F_\mathcal{Q}$ reaches* full reducer state *if the following conditions hold:*

1. $\mathbb{R}_k \overline{\ltimes} \widetilde{\mathbb{R}}_u = \emptyset$ *for the root of $\mathcal{T}_\mathcal{Q}$, R_k and its children R_u. The content of R_k satisfying the condition is denoted by $\mathbb{R}_k^*$;*
2. $\mathbb{R}_u \overline{\ltimes} \mathbb{R}_i^* = \emptyset$ *for all the leaf relations R_u of $\mathcal{T}_\mathcal{Q}$ and their parent R_i. The content of R_u satisfying the condition is denoted by $\mathbb{R}_u^*$. Furthermore, $\widetilde{\mathbb{R}}_u = \mathbb{R}_u$; and*

3. $(\mathbb{R}_i \overline{\ltimes} \widetilde{\mathbb{R}}_u) \cup (\mathbb{R}_i \overline{\ltimes} \mathbb{R}_j^*) = \emptyset$ *for internal° relation* R_i*, its child relations* R_u*, and its parent* R_j*. The content of* R_i *satisfying the condition is denoted by* $\mathbb{R}_i^*$*. If content of* R_i *satisfies* $\mathbb{R}_i \overline{\ltimes} \widetilde{\mathbb{R}}_u = \emptyset$ *only, we denote the content of* R_i *as* $\widetilde{\mathbb{R}}_i$*.*

▶ **Theorem 35.** *The cost of* $\mathcal{T}_\mathcal{Q}$ *under* YA *is*

$$\sum_{u=1}^{|\mathcal{A}|} |\mathbb{R}_u^*| \tag{13}$$

$$+ \sum_{i=1}^{s} (|\mathbb{R}_i^*| + \sum_{t=0}^{|\mathcal{B}^i|-1} |\mathbb{R}_i^{[t]} \ltimes \widetilde{\mathbb{R}}_u^{t+1}|) \tag{14}$$

$$+ \sum_{t=0}^{|\mathcal{C}|-1} |\mathbb{R}_k^{[t]} \ltimes \widetilde{\mathbb{R}}_u^{t+1}| \tag{15}$$

$$+ \sum_{i=k}^{1} |f(S_i)| \tag{16}$$

$$+ \sum_{i=2}^{k} |\mathbb{R}_i^*| \tag{17}$$

We define the following three sets over the relations in $\mathcal{T}_\mathcal{Q}$. First, $\mathcal{A}$ consists of all the leaf relations R_u. Second, $\mathcal{B}$ consists of R_u whose parents R_i are internal° relations. We partition $\mathcal{B}$ by the parent of R_us. Then, we have $\mathcal{B}^1, \ldots, \mathcal{B}^s$. $|\mathcal{B}^i|$ indicates the number of relations in $\mathcal{T}_\mathcal{Q}$ that are children of R_i. We label those relations $R_u^1, \ldots, R_u^{|\mathcal{B}^s|}$. Third, $\mathcal{C}$ comprises all the relations R_u that are children of R_k. The children of R_k are labeled $R_u^1, \ldots, R_u^{|\mathcal{C}|}$. Equation (18) defines $\mathbb{R}_i^{[t]}$, which reflects the gradual removal of dangling tuples of R_i during semijoins.

$$\mathbb{R}_i^{[t]} = \begin{cases} \mathbb{R}_i & \text{if } t = 0 \\ \mathbb{R}_i^{[t-1]} \ltimes \widetilde{\mathbb{R}}_u^t & \text{otherwise} \end{cases} \tag{18}$$

Suppose the join order (w.r.t. $\mathcal{T}_\mathcal{Q}$) determined either syntactically from $\mathcal{T}_\mathcal{Q}$ or from the DP algorithm is $[S_k, \ldots, S_1]$ where $S_j = R_i$ for some $i \in [k]$. $f(S_j)$ in Equation (19) computes the size of intermediate results that are part of the final join result.

$$f(S_i) = \begin{cases} \mathbb{S}_i^* & \text{if } i = k \\ \mathbb{S}_i^* \ltimes f(S_{i+1}) & \text{otherwise} \end{cases} \tag{19}$$

G.5 Empirical study of PT performance on the predicate transfer graph versus $\mathcal{T}_\mathcal{Q}$

Predicate transfer graph is a directed query graph. PT construct the predicate transfer graph from the query graph using a simple heuristic: for an edge of two relations, the head of the edge is the relation with bigger size. For star schema queries in our setup where R_k is the fact table, predicate transfer graph is identical to $\mathcal{T}_\mathcal{Q}$: query graph is identical to the undirected join tree and the heuristic applied on the query graph leads to $\mathcal{T}_\mathcal{Q}$. Thus, PT on the predicate transfer graph (denoted by PTO) has identical performance as PT on $\mathcal{T}_\mathcal{Q}$ on star schema queries. PTO can have a different performance compared with PT when one of the following conditions happen: (1) the query graph is not identical to an undirected join tree; (2) when undirected join tree and the query graph are identical, $\mathcal{T}_\mathcal{Q}$ created from

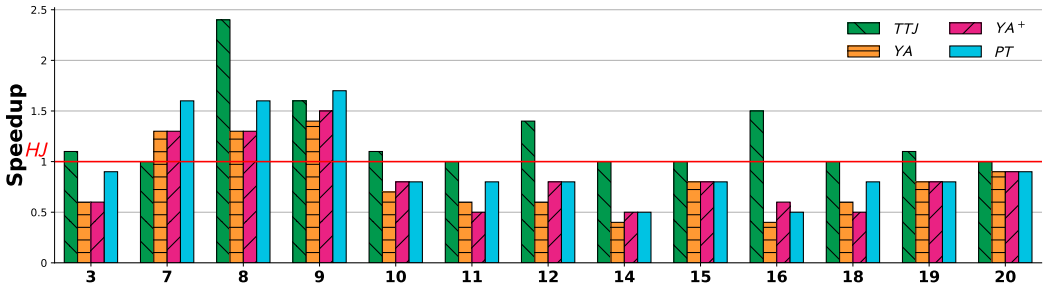

Figure 8 Speedup of TTJ, YA, YA$^+$, PT over HJ on 13 TPC-H queries

costing is different from the predicate transfer graph created by the heuristics; and (3) both $\mathcal{T}_\mathcal{Q}$ and the predicate transfer graph are identical but the order of passing Bloom filters are different.

Table 1 Speedup of PT and PTO compared with HJ on Q7 and Q8 in TPC-H

Method	Q7	Q8
PT	1.6×	1.6×
PTO	0.6×	0.7×

We empirically compare PT and PTO on Q7 and Q8 in TPC-H. The performance result is shown in Table 1. In Q7, condition (2) happens where PT and PTO have different tree structures. In Q8, condition (3) happens where both PT and PTO share the same $\mathcal{T}_\mathcal{Q}$ but Bloom filters are applied in different orders.

H TPC-H Results

Figure 8 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 four algorithms: the maximum speedup of YA, YA$^+$, and PT is 1.4×, 1.5×, 1.7×, respectively. TTJ has steady speedup over the benchmarked TPC-H queries with average (geometric mean) 1.2× compared with 0.69× from YA, 0.78× from YA$^+$, and 0.84× from PT. The minimum speedup of TTJ, YA, YA$^+$, and PT is 1.0×, 0.4×, 0.5×, 0.5×, respectively. From the aggregate statistics we can see: First, TTJ has more steady speedup than YA, YA$^+$, and PT on the entire workload: TTJ has higher average and minimum speedup than the other three algorithms. Second, YA, YA$^+$, and PT outperform TTJ 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 $\ltimes$ nation already removes more than 90% of tuples from supplier because |nation| = 1. The largely reduced supplier speeds up the subsequent semijoin lineitem $\ltimes$ 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 in a small amount of time.

I Measurement of the Number of Dangling Tuples Removed

Figures 9 and 10 empirically measure the amount of tuples removed by TTJ and all the algorithms we compared on TPC-H and SSB. From the figures we see that TTJ always remove the least amount of dangling tuples because TTJ reaches the clean state and the clean state requires the least amount of dangling tuples removed compared with YA and

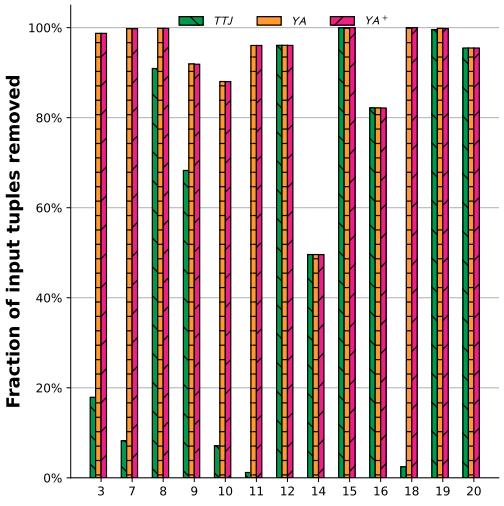

Figure 9 Fraction of tuples removed from the input relations by TTJ, YA, YA$^+$, and PT on TPC-H

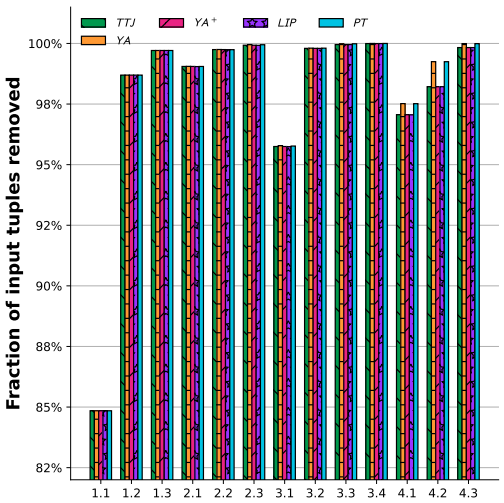

Figure 10 Fraction of tuples removed from the input relations by TTJ, YA, YA$^+$, LIP, and PT on SSB

YA$^+$ (Corollary 15). This confirms that TTJ, although intuitively combining the bottom-up semijoin pass with the join pass, is different from YA$^+$.

J Proof of YA$^+$

YA$^+$ states that bottom-up semijoin pass and top-down join pass over $\mathcal{T}_\mathcal{Q}$ is sufficient to provide $\mathcal{O}(n+r)$ guarantee. We formalize the statement into Theorem 36.

▶ **Theorem 36.** *Given a join tree $\mathcal{T}_\mathcal{Q}$ and root R_1, one can compute join with the following two steps:*

1. *apply bottom-up semijoin pass $HF_\mathcal{Q}$ on $\mathcal{T}_\mathcal{Q}$;*
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 $HF_\mathcal{Q}$, $R_1 = R_1^*$ (Lemma 4 of [12]) 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.

Proof. Proof by induction on the height of $\mathcal{T}_\mathcal{Q}$. Base case. Suppose the height of $\mathcal{T}_\mathcal{Q}$ is 0. Claim trivially holds. Suppose the claim holds for all queries whose height of $\mathcal{T}_\mathcal{Q} < h$. We want to show the claim holds for height of $\mathcal{T}_\mathcal{Q}$ 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 $\mathcal{T}_\mathcal{Q}$ 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 $HF_\mathcal{Q}$, $R_1 \ltimes (R_j \ltimes R_i)$ is executed. Because $\mathcal{T}_\mathcal{Q}$ 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 $P_{\mathcal{Q},1}$ 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. ◀

▶ **Corollary 37.** *The algorithm in Theorem 36 runs $\mathcal{O}(n + r)$, which is the same as* YA.

Corollary 37 immediately follows from Theorem 36 because intermediate result size is smaller than the final result size.

K Discussion and Related Work

We organize the related work in four categories. First, *CSP.* The equivalence between CQ evaluation and CSP is established by [38, 15]. TreeTracker in [9] solves a CSP for one solution without preprocessing the CSP. TTJ extends TreeTracker into query evaluation by (1) returning all possible solutions, and (2) blending the ideas from TreeTracker into physical operators in a query plan. Second, *Semijoin reduction.* An intensive research has been done on using semijoin to improve query evaluation speed [12, 59, 36, 13, 60, 40, 17, 66]. TTJ achieves a similar effect (clean state) as performing semijoin reduction without explicitly using semijoins. Third, *SIP.* `deleteDT()` of TTJ takes the form of SIP [68, 32, 35, 33, 8, 29, 42, 44, 22, 52, 54, 48, 23, 26, 67]. TTJ is different from the prior approaches in one or more of the following aspects: (1) TTJ does not introduce any preprocessing steps; (2) TTJ does not use Bloom filters, bitmaps, or semijoins; and (3) TTJ provides optimality guarantee. Third, *Worst-Case Optimal Join (WCOJ) algorithms.* A related line of work is to implement WCOJ algorithms efficiently [43, 6, 34, 25, 2, 64, 63]. TTJ is orthogonal to such direction as TTJ focuses on ACQ evaluation with query-specific output size r whereas WCOJ focuses on worst-case optimal join problem with worst-case output size bounded by AGM. In addition, comparing to WCOJ algorithms, which commonly use multi-way join operators, TTJ uses binary physical operators in iterator interface.