Yannakakis$^+$: Practical Acyclic Query Evaluation with Theoretical Guarantees

QICHEN WANG[*], Hong Kong Baptist University, Hong Kong SAR
BINGNAN CHEN[*], Hong Kong University of Science and Technology, Hong Kong SAR
BINYANG DAI, Hong Kong University of Science and Technology, Hong Kong SAR
KE YI, Hong Kong University of Science and Technology, Hong Kong SAR
FEIFEI LI, Alibaba Group, China
LIANG LIN, Alibaba Group, China

Acyclic conjunctive queries form the backbone of most analytical workloads, and have been extensively studied in the literature from both theoretical and practical angles. However, there is still a large divide between theory and practice. While the 40-year-old Yannakakis algorithm has strong theoretical running time guarantees, it has not been adopted in real systems due to its high hidden constant factor. In this paper, we strive to close this gap by proposing Yannakakis$^+$, an improved version of the Yannakakis algorithm, which is more practically efficient while preserving its theoretical guarantees. Our experiments demonstrate that Yannakakis$^+$ consistently outperforms the original Yannakakis algorithm by 2x to 5x across a wide range of queries and datasets.

Another nice feature of our new algorithm is that it generates a traditional DAG query plan consisting of standard relational operators, allowing Yannakakis$^+$ to be easily plugged into any standard SQL engine. Our system prototype currently supports four different SQL engines (DuckDB, PostgreSQL, SparkSQL, and AnalyticDB from Alibaba Cloud), and our experiments show that Yannakakis$^+$ is able to deliver better performance than their native query plans on 160 out of the 162 queries tested, with an average speedup of 2.41x and a maximum speedup of 47,059x.

CCS Concepts: • **Information systems → Query optimization**; **Query planning**.

Additional Key Words and Phrases: conjunctive query; acyclic joins; cost-based optimizer; query rewrite

ACM Reference Format:

Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis$^+$: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 235 (June 2025), 28 pages. https://doi.org/10.1145/3725423

Acknowledgments

This work was supported by Hong Kong RGC Grants (Project No. 12200524, 16205422, 16204223, 16203924, C2004-21GF, and C2003-23Y) and an AIR grant from Alibaba Cloud.

[*]Both authors contributed equally to this research.

Authors' Contact Information: Qichen Wang, qcwang@comp.hkbu.edu.hk, Hong Kong Baptist University, Hong Kong SAR; Bingnan Chen, bchenba@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong SAR; Binyang Dai, bdaiab@ust.hk, Hong Kong University of Science and Technology, Hong Kong SAR; Ke Yi, yike@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong SAR; Feifei Li, lifeifei@alibaba-inc.com, Alibaba Group, China; Liang Lin, yibo.ll@alibaba-inc.com, Alibaba Group, China.

ACM 2836-6573/2025/6-ART235
https://doi.org/10.1145/3725423

1 Introduction

Selection-join-projection-aggregation queries, a.k.a. *conjunctive queries (CQs)*, form the backbone of most analytical workloads[1]. The following query, which is a slightly simplified version of TPC-H Query 9 [6], is one such example:

```
SELECT n_name, o_orderkey, l_returnflag, SUM(ps_supplycost * l_quantity) AS part_cost
FROM nation, supplier, part, orders, lineitem, partsupp
WHERE o_orderdate < DATE '1996-12-31' and o_orderdate >
  DATE '1996-01-01' and p_name LIKE '%blue%'
  and o_orderkey = l_orderkey and ps_suppkey = l_suppkey
  and ps_partkey = l_partkey and p_partkey = l_partkey
  and s_suppkey = l_suppkey and s_nationkey = n_nationkey
GROUP BY n_name, o_orderkey, l_returnflag;
```

Due to their central importance, how to evaluate conjunctive queries efficiently has been extensively studied in the database community, from both practical and theoretical angles. The predominant approach, implemented in most relational engines, aims to find an optimal query plan that takes the form of a directed acyclic graph (DAG). The leaves of the DAG correspond to the input relations, while each internal node represents a relational operator, which can be either unary (selection, projection, and aggregation) or binary (join and semi-join), and the root node of the DAG yields the query results.

Example 1.1. We ran the query above in DuckDB, a popular column-based relational engine especially optimized for analytical workloads. The query plan it used is as follows (we rename the join attributes and use the natural join syntax):

(1) $J_1 \leftarrow \pi_{\text{partkey,p\_name}}\sigma_{\text{p\_name LIKE...}}(\texttt{part}) \bowtie \pi_{\text{partkey,orderkey,suppkey,l\_returnflag,l\_quantity}}(\texttt{lineitem})$;

(2) $J_2 \leftarrow \pi_{\text{orderkey}}\sigma_{\dots<\text{o\_orderdate}<\dots}(\texttt{orders}) \bowtie J_1$;

(3) $J_3 \leftarrow \pi_{\text{suppkey,nationkey}}\texttt{supplier} \bowtie \pi_{\text{n\_name,nationkey}}\texttt{nation}$;

(4) $J_4 \leftarrow J_3 \bowtie J_2$;

(5) $J_5 \leftarrow J_4 \bowtie \pi_{\text{partkey,suppkey,ps\_supplycost}}(\texttt{partsupp})$;

(6) $Q \leftarrow \gamma_{\text{n\_name,orderkey,l\_returnflag,SUM(ps\_supplycost*l\_quantity)}}(J_5)$.

However, from the theoretical angle, this query plan is sub-optimal for the following two reasons: First, there might be many *dangling tuples* that are unnecessarily involved in the joins, especially when the query has some highly selective predicates. For instance, the predicate on `o_orderdate` may filter out a large portion of the `orders` table, which means that many intermediate join results in J_1 will not be able to join with any tuple in the `orders` table, hence become *dangling*. These dangling tuples could blow up the intermediate join size to $O(N^\rho)$ in the worst case, where N is the input size and ρ is the *fractional edge cover number*[2] of the query. Second, this plan evaluates the full multi-way join before the aggregation. This can be sub-optimal since the full join size (denoted by F subsequently) can be much larger than the final query output size (denoted by M), which is equal to the number of groups. In particular, when the aggregation does not have a `GROUP BY` clause, it aggregates all join results into a single value, hence $M = 1$.

In practice, nevertheless, these two potential risks may not materialize because data is often "nice": We ran the query on the TPC-H benchmark dataset with a scale factor (SF) of 500 in DuckDB,

[1]The traditional definition of conjunctive queries does not consider aggregations. The incorporation of aggregations is introduced in [10, 42] under the semiring framework; see Section 2.1 for details.

[2]$\rho = 4$ for TPC-H Q9. Please see [14] for the precise definition of ρ, but this is not crucial for understanding the paper.

and it finished in just 9.2 seconds. In particular, this is because all the joins in this query are between a primary key (PK) and a foreign key (FK), which limits all intermediate join sizes, as well as the full join size F, to at most the input size N. To expose the risk, we removed the PK constraints and duplicated an SF-100 dataset 5 times. This results in a dataset of the same size, but each PK now has 5 copies. This turns the joins into many-to-many joins, and the intermediate join sizes are no longer bounded by N. On this dataset, DuckDB's running time blows up to 488 seconds, a 50x increase. We have also tested other benchmarks with naturally occurring many-to-many joins, such as LSQB [49] and JOB [45], and observed similar phenomenon (please see Section 7 for detailed results).

Back to the theory side, there is actually a 40-year-old solution that already addressed these issues when the query is *acyclic* (TPC-H Q9 is acyclic, and the formal definition will be given in Section 2). In 1981, Yannakakis [72] gave an algorithm that has a worst-case running time of $O(N + M)$ or $O(\min(NM, F))$, depending on whether the query has a certain property known as *free-connex* (detailed definition given in Section 2). Note that such running times are especially appealing when M is small, which is often the case for analytical queries that return aggregated results. Furthermore, the $O(N + M)$ time, which is achievable for free-connex queries, is clearly asymptotically optimal. Yannakakis' algorithm achieves these running times based on two key ideas: (1) use a series of semi-joins to remove all the dangling tuples before doing any joins, and (2) push the aggregations over joins as much as possible.

Unfortunately, despite its nice theoretical guarantees, Yannakakis' algorithm has not been adopted in any query engines due to its large hidden constant factor [54]. Indeed, we tested Yannakakis' algorithm in DuckDB on the TPC-H dataset with SF=500, and it took 21.3 seconds to evaluate Query 9, more than double that of DuckDB's query plan shown in Example 1.1. Similar results have also been observed in [29]. On the 5-copy dataset, however, we do see a significant improvement: Yannakakis' algorithm still runs in around 21 seconds (thanks to its worst-case guarantee), much faster than DuckDB's query plan which took 488 seconds.

1.1 Our Contributions

This paper presents Yannakakis^{+}, an improved version of the Yannakakis algorithm, with the following properties:

(1) It enjoys the same theoretical guarantee as the original Yannakakis algorithm on acyclic queries, i.e., it runs in $O(N + M)$ time if the query is free-connex, and $O(\min(NM, F))$ time otherwise.
(2) It is more practically efficient than the Yannakakis algorithm on both PK-FK joins and many-to-many joins. It consistently outperforms the Yannakakis algorithm by 2x to 5x (the maximum speedup is 87x) across four different SQL engines and a variety of queries/datasets. It thus covers the shortcomings of the Yannakakis algorithm on PK-FK joins, while extending its gain on many-to-many joins, as well as on queries that involve both types of joins. This makes Yannakakis^{+} the method of choice for a wide range of queries and datasets: Out of a total of 162 queries tested, Yannakakis^{+} is able to improve the SQL engines' own plans on 160 of them, with an average speedup of 2.41x and a maximum speedup of 47,059x.
(3) Yannakakis^{+} is also pure relational, in the sense that it can be formulated as a DAG query plan consisting of standard relational operators (see Table 1 for the operators that are needed). In fact, we were able to implement Yannakakis^{+} completely outside a SQL engine, by generating the query plan in the form of SQL statements. This allows Yannakakis^{+} to be used as a simple plug-in on top of any SQL engine, modulo minor changes in the syntax of the generated SQLs.

Operator	SQL Query	Complexity
Selection($\sigma_f(R)$)	`SELECT * FROM R WHERE f;`	$O(\lvert R\rvert)$
Projection($\pi_{\boldsymbol{E}}(R)$)	SELECT $\boldsymbol{E}$, $\oplus(v)$ AS v FROM R GROUP BY $\boldsymbol{E}$;	$O(\lvert R\rvert)$
Join($R_1 \bowtie R_2$)	SELECT *, $R_1.v \otimes R_2.v$ AS v FROM R_1 NATURAL JOIN R_2;	$O(\lvert R_1\rvert + \lvert R_2\rvert + \lvert R_1 \bowtie R_2\rvert)$
SemiJoin($R_1 \ltimes R_2$)	SELECT * FROM R_1 WHERE $R_1.key$ in (SELECT DISTINCT $R_2.key$ FROM R_2);	$O(\lvert R_1\rvert + \lvert R_2\rvert)$

Table 1. Summary of relation operators and the corresponding SQL queries, where v represents the annotation

Furthermore, as many other queries can be reduced to acyclic CQs, such as cyclic CQs, queries with conjunctive sub-queries, unions and differences of CQs, top-k queries, etc, Yannakakis$^+$ can also be used to improve their evaluation by combining with other techniques. We describe these extensions in Section 4.

Technical highlights. The practical improvements from Yannakakis to Yannakakis$^+$ are mostly driven by the following two observations. First, the original Yannakakis algorithm, due to its theoretical motivation, separates the evaluation process into two distinct stages: The first stage uses two passes of semi-joins to remove all the dangling tuples, which takes $O(N)$ time. Then the second stage uses a series of aggregation-joins to compute the query results, which takes $O(M)$ time (assuming the query is free-connex). While theoretically clean, this separation incurs unnecessary computational overheads. In Yannakakis$^+$, we push some aggregation-joins to before the semi-joins as much as possible, which is important since the aggregations can greatly reduce the data size, especially for queries with a small query output size M, while each join can remove a relation. Furthermore, we also reduce the number of semi-joins needed; in particular, for a class of queries known as *relation-dominated*, no semi-join is used at all. A possibly undesirable consequence of removing some of the semi-joins is that not all dangling tuples are removed, so a technical challenge in our development is to prove that the remaining dangling tuples do not affect the worst-case running time. In Section 3, we describe these changes that we make to the Yannakakis algorithm.

Second, both Yannakakis and Yannakakis$^+$ actually generate a family of query plans instead of a single one. Theoretically, all these plans have the same asymptotic running time, but they differ in the hidden constant. Thus, it is important to pick an optimal (or near-optimal) plan from this family. Towards this end, we design a query optimizer tailored for Yannakakis$^+$. Our optimizer follows the standard query optimization pipeline, consisting of a rule-based component and a cost-based component. However, we must introduce some changes to both components, since Yannakakis$^+$ has a different search space of query plans that existing query optimization methods do not cover. We describe our Yannakakis$^+$ optimizer in Section 5.

1.2 Related Work

Efficient evaluation of conjunctive queries has been extensively studied in the literature. Sideways information passing (SIP) [9] is a widely used technique for query optimization that reduces intermediate results, and it is adopted by systems such as DBMS X and Amazon Redshift [35]. However, unlike the Yannakakis algorithm, SIP does not remove all dangling tuples when the query contains more than two relations, which can lead to suboptimal plans. Meanwhile, worst-case optimal join algorithms (WCOJ) [55] perform better on highly cyclic queries. The Yannakakis algorithm and WCOJ can be combined using the generalized hypertree decomposition framework [30, 31] to provide running times that depend on the level of cyclicity of the query, measured by various *width parameters* [11, 30, 31, 33]. Recent studies extend the Yannakakis algorithm to support different operators and scenarios, including projections [15], aggregations [10, 42], unions [21], differences [40], comparisons [67], top-k queries [66], dynamic [41, 65] or secure [68] query processing. While these developments are promising, very few of them have made their way to real systems yet.

Several recent works [18, 19, 71, 75] focus on implementing the Yannakakis algorithm efficiently within a particular database engine, but they do not change the algorithm itself. In contrast, we have improved the algorithm. Thus, their techniques are complementary to ours and can be combined with our approach when Yannakakis$^+$ is integrated into their target database engine. Furthermore, Yannakakis$^+$ is aimed at conjunctive queries with (group-by) aggregations, while [18, 19, 71] only considers full joins. For cyclic queries, RelationalAI [57] and Umbra [28] adopt WCOJ inside databases, but they have to build the engine from ground up, since WCOJ does not directly generate a DAG plan using standard operators available in existing systems. Our method can also be combined with WCOJ to handle cyclic queries, as explained in Section 4. We have not implemented this combination, since we prefer a purely relational approach that yields standard DAG query plans.

We prove the worst-case running time of Yannakakis$^+$ based on the running times in Table 1. If indexes are available, some of these operators can be executed faster, e.g., selection can be sped up to $O(\log|R| + |\sigma_f(R)|)$ when there is a B-tree index and f is a range predicate; assuming $|R_2| > |R_1|$ and there is a hash-index on R_2, then the join can be computed in time $O(|R_1| + |R_1 \bowtie R_2|)$. There is an extensive literature on indexing techniques [22, 25, 38, 44]. The availability of indexes can only make Yannakakis$^+$ run faster, so all our theoretical guarantees are not affected; in practice, it can be factored into our cost-based optimizer to pick the best plan in the Yannakakis$^+$ family.

Cost-based optimization is an important step in reducing the hidden constant factor of query plans, which is also used in Yannakakis$^+$. It involves three main components: cardinality estimation (CE), cost model (CM), and plan enumeration (PE). CE employs data statistics and assumptions on data distribution to estimate tuple counts using synopsis-based (e.g., histogram-based [12, 43] and sketch-based [20, 58]), sampling-based [23, 63, 70, 74], and learning-based methods [39, 62, 69]. CM translates the database state (which relations are in memory, availability of indexes, etc.) and cardinality estimates into execution costs, with traditional models defined by experts and modern, adaptive learning-based methods [46, 47, 60]. PE identifies the query plan with minimal cost, employing both non-learning (dynamic programming [50, 51, 59], top-down strategies [24, 27]) and learning-based approaches [37, 48]. For CE and CM, we can use existing techniques. However, we have to design new PE methods, since Yannakakis$^+$ has a different search space.

2 Preliminaries

2.1 Conjunctive Queries

We consider *conjunctive queries (CQs)* of the following form:

$$Q = \pi_O \left(R_1(\mathcal{A}_1) \bowtie R_2(\mathcal{A}_2) \bowtie \cdots \bowtie R_n(\mathcal{A}_n)\right), \tag{1}$$

where each $R_i(\mathcal{A}_i)$ is a relation with a set of attributes $\mathcal{A}_i$, for $i = 1, 2 \ldots, n$. The same relation may appear more than once with attribute renamings (i.e., self-joins); we consider them as logical copies of the same relation. We use $\boldsymbol{R} = \{R_1, \cdots, R_n\}$ to denote the set of all relations in the query, and $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cup \cdots \cup \mathcal{A}_n$ the set of all attributes. For a subset of the relations $\mathcal{S} \subseteq \boldsymbol{R}$, let $\mathcal{A}(\mathcal{S})$ be the attributes that appear in $\mathcal{S}$; and define $\bar{\mathcal{A}}_i = \mathcal{A}(\boldsymbol{R} - \{R_i\})$, i.e., all attributes except those that only appear in R_i.

The original work of Yannakakis only considered a (distinct) projection π_O after the multi-way join. The extension to aggregations is made in [10, 42], who use semirings to formalize the types of aggregations that can be supported. Let $(\mathbb{S}, \oplus, \otimes)$ be a communicative semiring, where $\mathbb{S}$ is the ground set, with $\oplus$ and $\otimes$ being its "addition" and "multiplication", respectively. Each input tuple $t \in R_i$ is associated with an *annotation* $v_i(t) \in \mathbb{S}$. These annotations are propagated through the join and projection, as follows. The annotation for any tuple t in the join results $\mathcal{J} = R_1(\mathcal{A}_1) \bowtie R_2(\mathcal{A}_2) \bowtie \cdots \bowtie R_n(\mathcal{A}_n)$ is the $\otimes$-aggregate of all the tuples, one from each relation, that make up t:

$$v(t) := \bigotimes_{R_i(\mathcal{A}_i) \in Q} v_i(\pi_{\mathcal{A}_i} t).$$

Then π_O performs a $\oplus$-aggregation grouped by O, i.e., the annotation of each tuple t in the final query results is

$$v(t) := \bigoplus_{\forall t' \in \mathcal{J}, \pi_O t' = t} v'(t').$$

The attributes in O are called the *output attributes*. Specially, if $O = \emptyset$ and $\mathcal{J} \neq \emptyset$, then the query returns the empty tuple $\langle\rangle$ associated with an annotation that aggregates all the join results:

$$v(\langle\rangle) = \bigoplus_{\forall t' \in \mathcal{J}} v'(t').$$

If $O = \mathcal{A}$ (i.e., $Q = \mathcal{J}$), then the query is called a *full query*, which does not perform any $\oplus$-aggregation.

Such a conjunctive query with properly defined annotations is equivalent to the following SQL query:

```
SELECT O, ⊕(v1 ⊗ ··· ⊗ vn)
FROM R1 NATURAL JOIN ··· NATURAL JOIN Rn
GROUP BY O;
```

Example 2.1. TPC-H Query 9 in Section 1 can be represented as the following conjunctive query over the semiring $(\mathbb{R}, +, \cdot)$:

$$\begin{aligned} Q_1 = &\pi_{x_1,x_2,x_8} ((R_1(x_1, x_2, x_3, x_4) \bowtie R_2(x_2, x_5) \bowtie R_3(x_3, x_4) \\ &\bowtie R_4(x_3, x_6) \bowtie R_5(x_4, x_7) \bowtie R_6(x_7, x_8)) \end{aligned}$$

where $R_1, R_2, R_3, R_4, R_5, R_6$ correspond to the relations `lineitem`, `orders`, `partsupp`, `part`, `supplier`, and `nation`, respectively, while $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ correspond to the attributes `l_returnflag`, `orderkey`, `partkey`, `supplierkey`, `o_orderdate`, `p_name`, `nationkey`, `n_name`. Note that we have dropped unnecessary columns and renamed the join attributes to fit the natural join syntax. For all tuples in R_2, R_4, R_5, R_6, their annotations are set to 1. For each tuple $t \in R_3$, set $v(t) :=$ `ps_supplycost`; for each tuple $t \in R_1$, set $v(t) :=$ `l_quantity`.

We have also omitted the selection operators σ, which can always be pushed down to the input relations. They can be handled by a table scan or more efficiently by index retrieval if available, which are issues orthogonal to this work. □

Note that this semiring formulation unifies most cases of the aggregation operator γ into π. In particular, projection can be considered as a special case of aggregation on the boolean semiring $(\{\text{False}, \text{True}\}, \wedge, \vee)$, and all tuples in the database are assigned annotation True. By choosing the semiring and annotations appropriately, this formulation incorporates a variety of aggregation queries. For example, in addition to the commonly used $(\mathbb{R}, +, \cdot)$ in the above example, the semiring $(\mathbb{R}, \text{MAX}, +)$ allows us to compute aggregations like `MAX(ps_availqty − l_quantity)` by setting the annotations in R_3 to `ps_availqty` and the annotations in R_1 to `−l_quantity`.

We will also make use of some other standard relational operators including selection, union, semi-join, and order-by. These operators will not change the annotations of the tuples. In particular, a semi-join $R_1 \ltimes R_2$ returns all the tuples in R_1 that can join with at least one tuple in R_2. Here, the tuples in R_1 retain their annotations in R_1 after the semi-join, and the annotation in R_2 are irrelevant.

When analyzing the running time of an algorithm, we adopt the standard RAM model of computation and consider the *data complexity*, i.e., the query size is taken as a constant. We will measure the running time of a query evaluation algorithm using three parameters: the total input size $N = \sum_{i=1}^{n} |R_i|$, the query output size M, and the full join size $F = |R_1(\mathcal{A}_1) \bowtie \cdots \bowtie R_n(\mathcal{A}_n)|$. Note that $M = F$ for a full query, but M could be much smaller than F for a non-full query.

2.2 Classification of CQs

In the study of CQs, the following classes have been identified to bear complexity-theoretical significance:

Acyclic CQs. There are many equivalent definitions for acyclic CQs, and we adopt the one based on *join trees* [16, 26]. A CQ Q is acyclic if there exists a tree $\mathcal{T}$ satisfying the following properties: (1) the set of nodes in $\mathcal{T}$ have a one-to-one mapping to the set of relations in Q; and (2) for each attribute $x \in \mathcal{A}$, all nodes of $\mathcal{T}$ containing x form a connected subtree of $\mathcal{T}$. The join tree may not be unique; the GYO algorithm [32, 73] can be used to decide whether a given query Q is acyclic, and if yes, find all possible join trees for Q. Because of the one-to-one mapping between the relations and the tree nodes, we may use these two terms interchangeably on a fix join tree $\mathcal{T}$. For a node/relation $R_i(\mathcal{A}_i)$ in $\mathcal{T}$, we often use $R_p(\mathcal{A}_p)$ to denote its parent node, and C_i its children.

Note that the acyclicity of a query does not concern its output attributes, which are instead considered in the following sub-classes of acyclic CQs.

Free-connex CQs. A CQ Q is *free-connex* if both Q and $Q \bowtie [O]$ are acyclic, where $[O]$ denotes a relation with all output attributes [15]. This definition, however, is not easy to use in query evaluation, since Q and $Q \bowtie [O]$ have different join trees. In this paper, we use the following equivalent definition[3] that uses a single join tree.

Lemma 2.2. *A CQ Q is* free-connex *if and only if it has a join tree $\mathcal{T}$ with a subtree $\mathcal{T}_n$ containing the root node that satisfies two conditions: (1) $O \subseteq \mathcal{A}(\mathcal{T}_n)$, where $\mathcal{A}(\mathcal{T}_n)$ represents the set of all attributes present in $\mathcal{T}_n$, and (2) for any non-root node $R(\mathcal{A}) \in \mathcal{T}_n$ with parent $R_p(\mathcal{A}_p)$, $\mathcal{A} \cap \mathcal{A}_p \subseteq O$. Such a $\mathcal{T}$ is called a free-connex join tree of Q, and $\mathcal{T}_n$ is referred to as its connex subset.*

In addition, we identify another sub-class of queries:

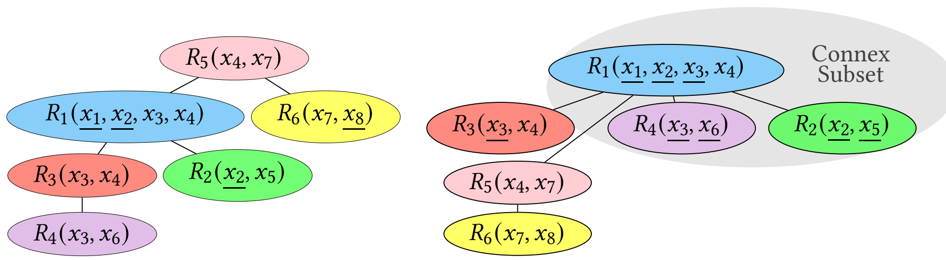
Relation-dominated CQs. A CQ Q is *relation-dominated* if Q is acyclic and there exists a relation $R_i(\mathcal{A}_i)$ such that $O \subseteq \mathcal{A}_i$. We call $R_i(\mathcal{A}_i)$ the *dominating relation*, and the join tree with $R_i(\mathcal{A}_i)$ as the root the relation-dominated join tree of Q. Note that for the special case $O = \emptyset$, the query is dominated by any of its relations, and any of its join trees is a relation-dominated join tree.

Example 2.3. TPC-H Query 9 (Q_1 in Example 2.1) is an acyclic query with two possible join trees $\mathcal{T}_1$ and $\mathcal{T}_2$ shown in Figure 1(a) and Figure 1(b).

Q_1 is not free-connex. But if we change the output attributes to $O = \{x_1, x_2, x_3, x_5, x_6\}$, then the resulting query

$$Q_2 \leftarrow \pi_{x_1,x_2,x_3,x_5,x_6} \left(\bowtie_{i \in [6]} R_i \right)$$

[3]Due to space constraints, all proofs are given in the technical report [64].



(a) Join tree $\mathcal{T}_1$ for Q_1

(b) Free-connex join tree $\mathcal{T}_2$ for Q_2, Q_3

Fig. 1. Two possible join trees for Q_1, Q_2 and Q_3. The output attributes are underlined.

is free-connex, with a free-connex join tree $\mathcal{T}_2$ shown in Figure 1(b). Note that $\mathcal{T}_1$ is not a valid free-connex join tree for Q_2 because the join attributes between R_1 and R_3 contain a non-output attribute x_4.

Furthermore, if we change the output attributes to $O = \{x_1\}$, then the query

$$Q_3 \leftarrow \pi_{x_1}\left(\bowtie_{i\in[6]} R_i\right)$$

is relation-dominated, by picking R_1 as the root of the join tree. □

2.3 The Yannakakis Algorithm

It is clear that all relation-dominated queries are free-connex queries, and all free-connex queries are acyclic queries. Acyclic and free-connex queries are at the core in the theory of query evaluation: All acyclic queries can be evaluated in $O(\min(NM, F))$ time [72], while free-connex queries can be evaluated in $O(N + M)$ time [15]. Both running times are achievable by the Yannakakis algorithm, which, on a given acyclic query Q with a join tree $\mathcal{T}$, works as follows:

(1) Traverse the tree in the post-order; for each visited tree node R_i and its parent node R_p, replace R_p with $R_p \ltimes R_i$;
(2) Traverse the tree in the pre-order; for each visited non-leave node R_i, for each $R_c \in C_i$, replace R_c with $R_c \ltimes R_i$;
(3) Traverse the tree in the post-order again; for each visited tree node R_i, replace R_p with $\left(\pi_{\mathcal{A}_p \cup O} R_i\right) \bowtie R_p$ and remove R_i from the tree.
(4) Until only one node R_r on the join tree, output $\pi_O R_r$ as the query result.

Example 2.4. Using the join tree $\mathcal{T}_1$ in Figure 1(a), the Yannakakis algorithm yields the following query plan for Q_1:

(1) $R_1 \leftarrow R_1 \ltimes R_2$;
(2) $R_3 \leftarrow R_3 \ltimes R_4$;
(3) $R_1 \leftarrow R_1 \ltimes R_3$;
(4) $R_5 \leftarrow R_5 \ltimes R_1$;
(5) $R_5 \leftarrow R_5 \ltimes R_6$;
(6) $R_6 \leftarrow R_6 \ltimes R_5$;
(7) $R_1 \leftarrow R_1 \ltimes R_5$;
(8) $R_3 \leftarrow R_3 \ltimes R_1$;
(9) $R_4 \leftarrow R_4 \ltimes R_3$;
(10) $R_2 \leftarrow R_2 \ltimes R_1$;
(11) $\mathcal{J}_1 \leftarrow \pi_{x_2} R_2 \bowtie R_1$;
(12) $\mathcal{J}_2 \leftarrow \pi_{x_3} R_4 \bowtie R_3$;
(13) $\mathcal{J}_3 \leftarrow \mathcal{J}_1 \bowtie \mathcal{J}_2$;
(14) $\mathcal{J}_4 \leftarrow \pi_{x_1,x_2,x_4} \mathcal{J}_3 \bowtie R_5$;
(15) $\mathcal{J}_5 \leftarrow \mathcal{J}_4 \bowtie R_6$;
(16) $Q_1 \leftarrow \pi_{x_1,x_2,x_8} \mathcal{J}_5$. □

A linear complexity $O(N + M)$ is clearly optimal; the optimality of the $O(\min(NM, F))$ bound is still elusive, but it is nevertheless the best-known worst-case running time for acyclic but non-free-connex CQs.

3 Yannakakis+

In this section, we describe Yannakakis+. For now, we assume that a join tree $\mathcal{T}$ (a free-connex join tree for a free-connex CQ, and a relation-dominated join tree for a relation-dominated query) is given; we will discuss how to pick a good join tree in Section 5.

3.1 First-round computation

Yannakakis+ consists of two rounds. The first round performs a post-order traversal on $\mathcal{T}$, as shown in Algorithm 1.

Algorithm 1: First round post-order traversal

Input: A join tree $\mathcal{T}$ for the acyclic query Q on relations $\boldsymbol{R}$
Output: A reduced join tree $\mathcal{T}'$ on relations $\boldsymbol{R}'$

Let $R_1, \cdots, R_n$ be arranged in some post-order of $\mathcal{T}$;
foreach $i \in [n-1]$ **do**
  Let $R_p(\mathcal{A}_p)$ be the parent node of $R_i(\mathcal{A}_i)$ on $\mathcal{T}$;
  if *R_i is a leaf node of $\mathcal{T}$ and $\mathcal{A}_i \cap O \subseteq \mathcal{A}_p$* **then**
    $R_p \leftarrow R_p \bowtie \left(\pi_{\mathcal{A}_p} R_i\right)$;
    $\mathcal{T} \leftarrow \mathcal{T} - \{R_i\}$, $\boldsymbol{R} \leftarrow \boldsymbol{R} - \{R_i\}$;
  else
    $R_i \leftarrow \pi_{O \cup \bar{\mathcal{A}}_i} R_i$;
    $R_p \leftarrow R_p \ltimes R_i$;
$R_n \leftarrow \pi_{O \cup \bar{\mathcal{A}}_n} R_n$;
return $\mathcal{T}, \boldsymbol{R}$

Below, we use three examples to illustrate how the first round works.

Example 3.1. First consider a simple query on two relations:

$$Q_4 \leftarrow \pi_{x_1}\left(R_1(x_1, x_2) \bowtie R_2(x_2, x_3)\right).$$

Note that this query is relation-dominated, hence also free-connex, and the (only) free-connex join tree has R_1 as the root and R_2 as the leaf.

The standard query plan used in most database systems for this query is

$$(1) \quad \mathcal{J} \leftarrow R_1 \bowtie R_2; \quad (2) \quad \text{return } T \leftarrow \pi_{x_1}\mathcal{J}.$$

This plan takes $O(N + F)$ time; recall that $F = |R_1 \bowtie R_2|$ is the full join size.

In contrast, the Yannakakis algorithm for this query achieves $O(N + M) = O(N)$ time (since $M \le N$ on this query), through the following plan:

$$\begin{array}{llll} (1) & R_1 \leftarrow R_1 \ltimes R_2; & (2) & R_2 \leftarrow R_2 \ltimes R_1; \\ (3) & R_1 \leftarrow R_1 \bowtie \pi_{x_2} R_2; & (4) & \text{return } T \leftarrow \pi_{x_1} R_1. \end{array}$$

Algorithm 1 on this query yields the following plan:

$$(1) \quad R_1 \leftarrow R_1 \bowtie \pi_{x_2} R_2; \quad (2) \quad \text{return } T \leftarrow \pi_{x_1} R_1.$$

Because only one relation remains after the first-round computation, Yannakakis$^+$ terminates without needing to do the second round. We see that the Yannakakis$^+$ plan is actually the same as the last two steps in the Yannakakis plan. Essentially, the observation is that, for this query, the two semi-joins are not necessary; doing the last two steps directly, even with the presence of dangling tuples, still guarantees $O(N)$ time, which we will prove more formally and generally later.

We ran the three query plans in DuckDB on the Epinion graph, where both R_1 and R_2 refer to the edge relation with 508,837 edges (namely, it is a self-join). We use the $(\mathbb{N}, +, \cdot)$ smearing and set all input tuples' annotations to 1, so the query returns the number of length-2 paths for each vertex x_1. The standard plan took 0.507 s, the Yannakakis plan took 0.243 s, while our new plan took 0.0366 s. □

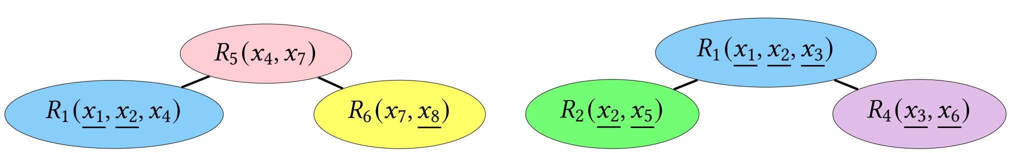
Example 3.2. Next, consider Q_2 from Example 2.3, which is free-connex but not relation-dominated. This query has more than one free-connex join tree; in this example, we use $\mathcal{T}_2$ in Figure 1(b). Then Algorithm 1 yields the following steps:

$$\begin{array}{llll} (1) & R_5 \leftarrow R_5 \bowtie \pi_{x_7} R_6; & (2) & R_1 \leftarrow R_1 \bowtie R_3; \\ (3) & R_1 \leftarrow R_1 \bowtie \pi_{x_4} R_5; & (4) & R_1 \leftarrow R_1 \ltimes R_2; \\ (5) & R_1 \leftarrow R_1 \ltimes R_4; & (6) & R_1 \leftarrow \pi_{x_1,x_2,x_3} R_1. \end{array}$$

The first three steps fall into the **if** part, since the output attributes in R_3, R_5, R_6 also appear in their parents. We do early aggregation and join for these relations, which are then removed. Steps (4)–(5) take the **else** part that does the semi-joins. Note that line 8 in Algorithm 1 is a no-op in this example. Step (6) performs the final aggregation of line 10 in Algorithm 1. We see Algorithm 1 has reduced the query to a full join (which we will show is true for all free-connex queries):

$$Q_2' \leftarrow R_1(x_1, x_2, x_3) \bowtie R_2(x_2, x_5) \bowtie R_4(x_3, x_6),$$

and the reduced join tree $\mathcal{T}_2'$ is shown in Figure 2(b). □



(a) Join tree $\mathcal{T}_1'$ for Q_1' after the first round. (b) Join tree $\mathcal{T}_2'$ for Q_2' after the first round.

Fig. 2. Two Jointrees for Q_1' and Q_2'.

Example 3.3. Finally, consider a non-free-connex query, Q_1 from Example 2.1. Suppose we use the join tree $\mathcal{T}_1$ in Figure 1(a). Then Algorithm 1 yields the following query plan:

$$\begin{array}{llll} (1) & R_1 \leftarrow R_1 \bowtie \pi_{x_2} R_2; & (2) & R_3 \leftarrow R_3 \bowtie \pi_{x_3} R_4; \\ (3) & R_1 \leftarrow R_1 \bowtie R_3; & (4) & R_1 \leftarrow \pi_{x_1,x_2,x_4} R_1; \\ (5) & R_5 \leftarrow R_5 \ltimes R_1; & (6) & R_5 \leftarrow R_5 \ltimes R_6; \end{array}$$

and the reduced query is

$$Q_1' \leftarrow \pi_{x_1,x_2,x_8} R_1(x_1, x_2, x_4) \bowtie R_5(x_4, x_7) \bowtie R_6(x_7, x_8)$$

with the join tree $\mathcal{T}_1'$ in Figure 2(a).

Compared with the free-connex Q_2, some non-output attributes of Q_1 remain, but Algorithm 1 did the best it can: The remaining attributes are either output attributes or join attributes (e.g., x_4 and x_7) that are "shielded" by the output attributes from below. □

Correctness. We will prove that the reduced query after the first round is equivalent to the original query.

Lemma 3.4. *On any acyclic query Q and its join tree $\mathcal{T}$, Algorithm 1 produces a query Q' that is equivalent to Q.*

Running time. We see that all the operators in the first-round computation has running time $O(N)$, and none of them increases the data size. This is clearly the case for all semi-joins and aggregations. Join is the only operator that may take more than linear time and enlarge the data size, but the join done in line 5 of Algorithm 1 is between R_p and $\pi_{\mathcal{A}_p} R_i$, and the latter's attribute set is a subset of the former. This is thus essentially a semi-join if annotations are not concerned. Since we are removing R_i, we need to use a join here to make sure that the annotations in $\pi_{\mathcal{A}_p} R_i$ are correctly multiplied by those in R_p.

Lemma 3.5. *The worst-case running time of Algorithm 1 is $O(N)$.*

Properties of the reduced query. In addition to being equivalent to the original query, we can prove the following properties of the reduced query Q', which will be useful in the second round:

Lemma 3.6. *For a given acyclic query Q, Algorithm 1 returns a reduced query Q' that only has*

(1) output attributes and join attributes;
(2) output attributes (i.e., Q' is a full query) if Q is free-connex;
(3) one relation consisting of only output attributes if Q is relation-dominated.

Combining Lemma 3.5 and Lemma 3.6(3), we obtain an algorithm for relation-dominated queries.

Theorem 3.7. *Algorithm 1 computes any relation-dominated query in $O(N)$ time.*

For other queries, we proceed to the second round.

3.2 Second-round computation

The second-round computation relies on the notion of *dangling-free* relations and *reducible* relations.

Definition 3.8 (Dangling-free Relations). Given a conjunctive query $Q := \pi_O\left(\bowtie_{k\in[n]} R_k\right)$, a relation R_i is *dangling-free* if on every database instance, for every $t \in R_i$, there exists a full join result $t' \in \bowtie_{k\in[n]} R_k$ such that $t = \pi_{\mathcal{A}_i} t'$.

Lemma 3.9. *For any acyclic query Q and any join tree $\mathcal{T}$ of Q, the root node R_r of $\mathcal{T}$ after the first round is dangling-free.*

Definition 3.10 (Reducible Relations). Let Q be an acyclic CQ and $\mathcal{T}$ be a join tree of Q. Consider a relation $R_i(\mathcal{A}_i) \in \mathcal{T}$, and let $R_j(\mathcal{A}_j)$ be a neighbor of R_i. We say that R_j is *reducible* for R_i if, for every other neighbor $R_k(\mathcal{A}_k)$ of R_i, $\mathcal{A}_k \cap \mathcal{A}_i \subseteq O$.

As example, in the join tree of Figure 1(b), R_1 has only one reducible relation R_3; R_4 are R_5 are not reducible for R_1, because R_3 and R_1 have a non-output join attribute x_4.

The following two special cases will be useful later: (1) For any leaf node, its parent is always reducible for it because it has no other neighbors. (2) In a full query, each node is reducible for all of its neighbors.

The second round revolves around dangling-free relations and their reducibles, since their joins have bounded size:

Lemma 3.11. *Given an acyclic query Q and a join tree $\mathcal{T}$ for Q after the first round, let R_i be a dangling-free relation and R_j be a reducible relation for R_i. Then $|R_i \bowtie R_j| = O(\min(NM, F))$. Furthermore, if $\bar{\mathcal{A}}_j \cap \mathcal{A}_j \subseteq O$, then $|R_i \bowtie R_j| = O(M)$.*

In the second round, we iteratively identify any dangling-free relation R_i and one of its reducible relations R_j, perform a join followed by a projection, and reduce the join tree by one relation, as shown in Algorithm 2. In the algorithm, Δ represents symmetric difference, i.e., $\mathcal{A}_i \Delta \mathcal{A}_j = (\mathcal{A}_i - \mathcal{A}_j) \cup (\mathcal{A}_j - \mathcal{A}_i)$.

Algorithm 2: Reduction($Q, \mathcal{T}, R_i, R_j$)

Input: A join tree $\mathcal{T}$ for the acyclic query Q with relation $\boldsymbol{R}$, where R_j is a reducible relation of a dangling-free relation R_i

Output: A resulting join tree $\mathcal{T}'$ and a reduced query Q' on relations $\boldsymbol{R}'$, where $|\boldsymbol{R}'| = |\boldsymbol{R}| - 1$

```
T' ← T, R' ← R;
R'_i ← π_{O∪(A_i Δ A_j)} (R_i ⋈ R_j);
R' ← (R' − {R_i} − {R_j}) ∪ {R'_i};
In T, merge R_i and R_j into R'_i;
Q' := π_O (⋈_{R∈R'} R);
return Q', T', R'
```

$\mathcal{T}' \leftarrow \mathcal{T}$, $\boldsymbol{R}' \leftarrow \boldsymbol{R}$;
$R'_i \leftarrow \pi_{O \cup (\mathcal{A}_i \Delta \mathcal{A}_j)} \left(R_i \bowtie R_j\right)$;
$\boldsymbol{R}' \leftarrow \left(\boldsymbol{R}' - \{R_i\} - \{R_j\}\right) \cup \{R'_i\}$;
In $\mathcal{T}$, merge R_i and R_j into R'_i;
$Q' := \pi_O \left(\bowtie_{R \in \boldsymbol{R}'} R\right)$;
return $Q', \mathcal{T}', \boldsymbol{R}'$

We need to show that a pair of dangling-free and reducible relations always exist, so that we can repeatedly apply Algorithm 2. It is easy to show that dangling-free relations always exist. In particular, the root of the join tree after the first round must be dangling-free (Lemma 3.9). Also, the join of a dangling-free relation with any other relation must still be dangling-free, so the newly generated relation R'_i by Algorithm 2 is also dangling-free.

However, it is not clear if reducible relations always exist. We consider the following two cases separately.

Free-connex queries. If Q is free-connex, then the query after the first round is full. By the observation earlier, every relation is reducible to all its neighbors. Thus we can apply Algorithm 2 on the root R_r and any of its child R_j. The newly generated relation is still dangling-free and it becomes the new root. We can thus repeatedly apply Algorithm 2 until only one relation remains.

In terms of running time, observe that in a full query, the second part of Lemma 3.11 applies, so the cost of each join is $O(N + M)$. Combining with Lemma 3.5, we conclude:

Theorem 3.12. *Algorithm 1 and 2 compute any free-connex query in $O(N + M)$ time.*

Example 3.13. We continue Example 3.2. After the first-round computation, the join tree is shown in Figure 2(b), which is a full query. The root R_1 is dangling-free, and both of its children R_2 and R_4 are reducible. Applying Algorithm 2 twice yields the following query plan (continuing the plan in Example 3.2):

$$(7) \quad R_1 \leftarrow R_1 \bowtie R_2; \quad (8) \quad Q \leftarrow R_1 \bowtie R_4. \quad \square$$

Non-free-connex queries. Although dangling-free relations must exist for non-free-connex queries after the first round (at least, the root of $\mathcal{T}$ is one), but they may not have any reducible neighbors. In this case, we use semi-joins to make additional relations dangling-free, based on the following lemma:

Lemma 3.14. *For any acyclic query Q, let $\mathcal{T}$ be the join tree after the first-round computation. Let R_i be any dangling-free relation in $\mathcal{T}$, and R_j be any child of R_i. If we replace R_j with $R'_j := R_j \ltimes R_i$, then the query is equivalent while R'_j is dangling-free for Q.*

As observed earlier, when a leaf becomes dangling-free, its parent must be reducible, so this strategy can always succeed in finding a pair of relations to apply Algorithm 2.

Example 3.15. We continue Example 3.3. The join tree is shown in Figure 2(a) after the first round. The root R_5 is dangling-free, but neither of its children is reducible. Then we can use a semi-join to make R_6 dangling-free, and then apply Algorithm 2 to merge R_5 and R_6. After this, R_1 becomes the only neighbor of R_5, hence reducible. The query plan is (continuing the plan in Example 3.3):

$$(7)\quad R_6 \leftarrow R_6 \ltimes R_5; \qquad (8)\quad R_5 \leftarrow \pi_{x_4,x_8}\left(R_5 \bowtie R_6\right);$$
$$(9)\quad Q_1 \leftarrow \pi_{x_1,x_2,x_8}\left(R_5 \bowtie R_1\right).$$

Compared with the original Yannakakis plan (Example 2.4), we see that our plan uses only 3 semi-joins as opposed to 10, and 3 aggregation-join operations have been pushed to before the semi-joins. We ran the three plans in DuckDB on the 5-copy SF=100 TPC-H dataset, DuckDB's plan took 488 s, the original Yannakakis plan took 21.1 s, while our new plan took 13.2 s. □

Finally, we can show that our algorithm achieves the same running time guarantee as that of the Yannakakis algorithm for acyclic but non-free-connex queries:

Theorem 3.16. *Algorithm 1 and 2 compute any acyclic query in $O(\min(NM, F))$ time.*

4 General Queries

4.1 Cyclic Queries

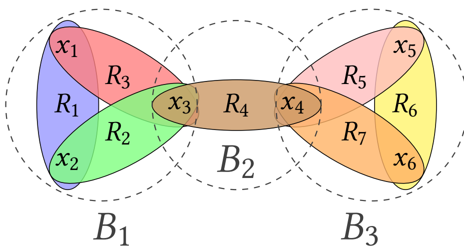
Our previous discussions were based on acyclic CQs with a join tree. For cyclic CQs, **Generalized Hypertree Decomposition (GHD)** [11, 30] is a powerful tool for efficiently transforming them into acyclic CQs. A GHD also takes the form of a tree $\mathcal{T}$, whose nodes are often called *bags*. But unlike the join tree for acyclic queries that maps each node to a single relation, each node Bag_j of $\mathcal{T}$ maps to a set of attributes $\mathcal{B}_j$, where (1) for every relation $R_i(\mathcal{A}_i)$, there exists a node Bag_j such that $\mathcal{A}_i \subseteq \mathcal{B}_j$ and (2) for each attribute x, all nodes of $\mathcal{T}$ containing x form a connected subtree of $\mathcal{T}$. Such a tree $\mathcal{T}$ is called a *generalized join tree*, and we said the tree is *generalized free-connex join tree* if it also satisfies the free-connex condition. Each bag $[\mathcal{B}]$ can be materialized by the following query:

$$Q_{\mathcal{B}} \leftarrow \bowtie_{R(\mathcal{A})\in \boldsymbol{R}, \mathcal{A}\subseteq\mathcal{B}\neq\emptyset} (R)\,. \tag{2}$$

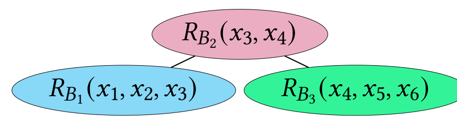
It should be noted that each relation can appear in multiple bags. In order to prevent miscalculations of the aggregate value, we create a special relation $R_i^{\mathbf{1}}$ for each $R_i \in \mathcal{R}$. For each $t \in R_i$, we add t to $R_i^{\mathbf{1}}$ with the annotation $v(t) = \mathbf{1}$. Then, we replace R_i with $R_i^{\mathbf{1}}$ for all bags except for one with $\mathcal{A}_i \subseteq \mathcal{B}$.

In order to evaluate a CQ Q on the given generalized join tree $\mathcal{T}$, we start by materializing each bag $[\mathcal{B}]$. This involves evaluating $Q_{\mathcal{B}}$ directly in the database with a binary join plan (or WCOJ if available) and then replacing the bag with the materialized relation $R_{\mathcal{B}}$. Our cost-based optimization further improves the pre-processing by selecting the best join orders for the binary join plan. Once this process is complete, the resulting tree becomes a normal join tree and can be evaluated directly using Yannakakis+.

In this work, we adopt a similar approach to the previous state-of-the-art [8], which exhaustively explores all possible generalized hypertree decompositions (GHDs). Our cost-based optimizer enhances the efficiency of GHD search by employing our cost estimator to obtain more accurate results than the standard search algorithms that rely on heuristics. Additionally, when calculating the size of each GHD bag, we take cardinality constraints into account. For example, if a bag contains relations $R_1(x_1, x_2)$ and $R_2(\underline{x_2}, x_3)$, where x_2 is a primary key for R_2, we conceptually merge them into a new relation $R_{12}(\overline{x_1}, x_2, x_3)$ with $|R_{12}| = |R_1|$. This approach provides a more accurate cost estimation, allowing our optimizer to select the most efficient GHD.



(a) Generalized Hypertree Decomposition



(b) Join tree

Fig. 3. An Example of GHD and its acyclic CQ.

Example 4.1. See Figure 3(a) as an example of GHD on a natural join of 7 relations: $R_1(x_1, x_2)$, $R_2(x_2, x_3)$, $R_3(x_3, x_1)$, $R_4(x_3, x_4)$, $R_5(x_4, x_5)$, $R_6(x_5, x_6)$ and $R_7(x_6, x_4)$. There are three bags in the decomposition:

$$R_{\mathcal{B}_1} \leftarrow R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_1);$$

$$R_{\mathcal{B}_2} \leftarrow R_4(x_3, x_4);$$

$$R_{\mathcal{B}_3} \leftarrow R_5(x_4, x_5) \bowtie R_6(x_5, x_6) \bowtie R_7(x_6, x_4).$$

After performing two triangle joins ($R_{\mathcal{B}_1}$ and $R_{\mathcal{B}_3}$) we get an acyclic join (line-3 join) as Figure 3(b). The tree can be evaluated in a total time of $O(N^{1.5}+M)$ with worst-case optimal joins, or $O(N^2+M)$ in most industrial database systems.

4.2 Sub-queries, Unions, Differences, and Top-k

The support for other operations in DBMS on our newly developed algorithm is natural by considering the underlying conjunctive query as a special relation. The evaluation and materialization of the underlying conjunctive query can be done by using the new algorithm. Then, additional operators can be applied to the query results by replacing the conjunctive queries with the new relation.

Example 4.2. Consider the TPC-H Benchmark Query 17:

```sql
SELECT SUM(l_extendedprice) / 7.0 as avg_yearly
FROM lineitem, part
WHERE p_partkey = l_partkey and p_brand = 'Brand#23' and p_container = 'MED BOX' and
     l_quantity < (
    SELECT 0.2 * avg(l_quantity) FROM lineitem WHERE l_partkey = p_partkey);
```

which is a nested query. To evaluate the nested query, our framework will first evaluate the underlying conjunctive query

```sql
SELECT 0.2* avg(l_quantity) as cnt FROM lineitem, part
WHERE p_partkey = l_partkey and p_brand = 'Brand#23' and p_container = 'MED BOX';
```

then using the query result R_Q as a new input relation, and evaluate another conjunctive query

```
SELECT SUM(l_extendedprice) / 7.0 as avg_yearly
FROM lineitem, part, R_Q
WHERE p_partkey = l_partkey and p_brand = 'Brand#23' and p_container = 'MED BOX' and
    l_quantity < cnt;
```

While this allows our algorithm to support universal SQL queries and our algorithm can guarantee an output-sensitive running time when evaluating the conjunctive queries, we cannot guarantee an output-sensitive running time for the entire query as the output size of these conjunctive queries can be significantly larger than the final size of the query results. Recent advances have shown that we can push down unions[21], differences[40], and Top-k[66] while evaluating those conjunctive queries. As a natural aspect of relational algorithms, our framework can be extended to support all these queries with additional rewrite steps and only add a constant or logarithmic cost to the complexity.

Example 4.3. Consider the following difference of conjunctive query (DCQ) studied in [40]:

$$\pi_{x_4}\left(R_1(x_1,x_2) \bowtie R_2(x_2,x_3,x_4) - R_3(x_1,x_2,x_3) \bowtie R_4(x_3,x_4)\right),$$

One way to evaluate the query is by first evaluating the two queries, $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$, and then calculating the difference between the two queries and performing the projection. However, it is possible that the final result is empty, even though the two conjunctive queries can produce $O(N^2)$ results in the worst case. By using the techniques introduced in [40], we can rewrite the process as follows:

$$\begin{aligned}&\pi_{x_4}\left(R_1 \bowtie R_2 - R_3 \bowtie R_4\right)\\ =&\pi_{x_4}\left(\left(R_1 - \pi_{x_1,x_2}R_3\right) \bowtie R_2\right) \cup \pi_{x_4}\left(R_1 \bowtie \left(R_2 - \left(\pi_{x_2,x_3}R_3\right) \bowtie R_4\right)\right),\end{aligned}$$

and

$$\begin{aligned}&\left(R_2 - \left(\pi_{x_2,x_3}R_3\right) \bowtie R_4\right)\\ =&R_2 \ltimes \left(\pi_{x_2,x_3}R_2 - \pi_{x_2,x_3}R_3\right) \cup R_2 \ltimes \left(\pi_{x_3,x_4}R_2 - R_4\right)\end{aligned}$$

where each individual query's output size is bounded by the actual output size of the DCQ. With Yannakakis+, those queries can be evaluated in $O(N + M)$ time, where M represents the actual output of the DCQ.

5 Query Optimization

Yannakakis+ provides the same asymptotic running time guarantee with any valid join tree (free-connex join tree, or relation-dominated join tree, respectively). However, there are still constant-factor differences between these join trees; even for the same join tree, different reduction orders during the two rounds of computations can also make some differences.

Example 5.1. Consider Q_1 from Example 2.1, where the new query plan, using the join tree $\mathcal{T}_1$, significantly improves the performance compared with the original query plan in DuckDB. However, our query optimizer can find a better join tree $\mathcal{T}_3$ by simply rotating the tree with R_1 as the root node, as shown in Figure 5. Despite the small changes to the join tree, the new query plan reduces the total intermediate results from 556,473,531 to 242,661,000 and eliminates one semi-join step in the second round of computation. The resulting running time on $\mathcal{T}_3$ is 6.800 s, which is approximately 49% less compared to the plan on $\mathcal{T}_1$. □

Thus, it is practically important to choose an optimal (or near-optimal) query plan from this family of plans. We have designed a query optimizer tailored for Yannakakis+, which consists of a rule-based component and a cost-based component, described below.

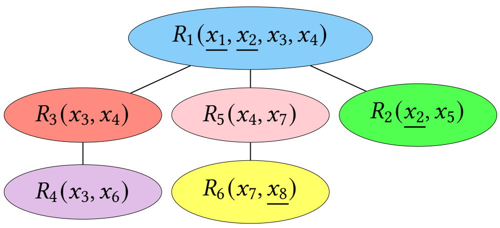


Fig. 5. Join tree $\mathcal{T}_3$ for Q_1.

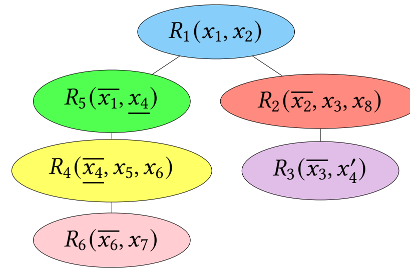


Fig. 6. Join tree for Q'_5.

5.1 Rule-Based Optimization

Cycle Elimination. Yannakakis$^+$ is designed to process acyclic queries and use GHD to transform cyclic queries into acyclic with extra cost. However, some queries, although cyclic, can be turned into acyclic without affecting the running time by exploiting the PK constraints.

Example 5.2. TPC-H query 5 can be represented as the following conjunctive query

$$Q_5 \leftarrow \pi_{x_5} R_1(x_1, x_2) \bowtie R_2(\bar{x}_2, x_3, x_8) \bowtie R_3(\bar{x}_3, x_4) \\ \bowtie R_4(\bar{x}_4, x_5, x_6) \bowtie R_5(\bar{x}_1, x_4) \bowtie R_6(\bar{x}_6, x_7),$$

where all primary keys are marked as $\bar{x}$. The query is not acyclic due to the cycle created by R_1, R_2, R_3, R_5. We break the cycle by renaming one of the x_4's into x'_4, but then reinforcing it by a selection:

$$Q'_5 \leftarrow \sigma_{x_4=x'_4}\Big(\pi_{x_5,x_4,x'_4} R_1(x_1, x_2) \bowtie R_2(\bar{x}_2, x_3, x_8) \bowtie R_3(\bar{x}_3, x'_4) \\ \bowtie R_4(\bar{x}_4, x_5, x_6) \bowtie R_5(\bar{x}_1, x_4) \bowtie R_6(\bar{x}_6, x_7)\Big).$$

Now the query (before the selection $\sigma_{x_4=x'_4}$) is acyclic, with a join tree shown in Figure 6. Meanwhile, since all joins are PK-FK joins, all intermediate join sizes are bounded by $O(N)$, so the overall running time is still $O(N)$, including the last selection step $\sigma_{x_4=x'_4}$. □

Aggregation Elimination. The PK constraint (in fact, any UNIQUE constraint) can help remove some redundant aggregations. When the group-by attribute is a PK, the aggregation can be eliminated, i.e., line 5 and 9 in Algorithm 1.

Semi-join Elimination. For a leaf relation R and its parent node R_p on the join tree, if the join key is a primary key of R and foreign key of R_p, and there is no filtering condition on R, then the semi-join step between R and R_p can be ignored. This is because the PK-FK relationship already ensures that all tuples can be joined.

Example 5.3. Consider the query Q_1 from Example 3.3. In the first round, R_2 was projected to x_2 before joining with R_1 to avoid duplication. However, if x_2 is a primary key of R_2, the projection is unnecessary. In addition, if the PK-FK relationship holds between R_5 and R_6, the semi-join in Step (1) and Step (7) can be omitted without increasing the complexity.

Pruning for Annotation. In Section 2, the definition of conjunctive queries requires an additional annotation column for each relation to support the calculation of aggregation functions. This helps to generalize the definition to accommodate various aggregations. However, in some cases, this annotation may be redundant, but our optimizer is designed to identify these cases and avoid the additional cost. Our experiments demonstrate the importance of this optimization for database systems with column-store.

Example 5.4. Consider the query from Example 2.1. If we change the corresponding aggregation function from SUM to MAX on ps_supplycost $*$ l_quantity to obtain the maximum cost, the query will be defined over the semiring $(\mathbb{R}, \max, \cdot)$. In this case, we won't need to assign additional annotations on relations except for Partsupp and Lineitem. Our optimizer detects such situations and eliminates those annotations from our plan.

Fusion of Dimension Relations. When a query involves joins between a large relation and multiple small relations, the optimizer can enhance efficiency by first join the small relations, or even using Cartesian products if they lack common attributes. This is because join or semi-join with the large relation can be more costly than performing a Cartesian product of the small relations. For example, in the query $R_1(x_1) \bowtie R_2(x_1, x_2) \bowtie R_3(x_2)$, if $|R_1|$ and $|R_3|$ are significantly smaller than $|R_2|$, we first perform the Cartesian product $R_1 \times R_3$. Then, we apply our new query plan, which saves one join or semi-join with the large relation R_2.

5.2 Cost-Based Optimization

Cost-based optimization in database systems is a key technique for enhancing query performance and resource usage. Our new algorithms specifically focus on the efficiency of a query plan within the algebraic structure. For all valid join trees, they have the same theoretical worst-case complexity. Therefore, it's important for us to take into account instance-specific information in order to identify the best query plan among all available options. In contrast to the standard binary join approach, which may not perform well due to the amplification of errors by join operations, operators like semi-join have a bounded cost that does not amplify errors. Additionally, the linear time guarantee provides an upper bound on the cost estimation. These factors make the standard cost-based optimization more effective for our new query plans.

Plan Enumeration (PE). The first step in plan enumeration is to generate all valid join trees for the given query. For acyclic queries, we use GYO reduction [32, 73] to enumerate all valid join trees. However, for cyclic hypergraphs, directly applying GYO reduction cannot reduce the query to an empty graph. Therefore, we compute all possible generalized hypertree decompositions (GHDs) [31].

After generating a set of valid plans, we employ the following pruning strategies to control their number:

- For queries with output attributes, we require the root node to contain output attributes;
- We prefer plans where the larger relations are at the top of the tree;
- Unlike current database optimizers that tend to favor left-deep plans, we prioritize bushy plans with lower heights.

These rules help avoid additional costs when propagating large relations through intermediate results and make it easier for child nodes to prune their parent nodes.

Cardinality Estimation (CE) and Cost Model (CM). Estimating the cardinality of intermediate results in a query plan has been extensively studied in the literature. Thanks to their theoretical guarantee, the Yannakakis$^+$ plans are less sensitive to CE/CM than traditional plans. Bad CE/CM leads to at most a constant-factor difference for Yannakakis$^+$, while they may incur a polynomial-factor degradation for traditional plans, from $O(N)$ to $O(N^2)$ or even worse. We used the standard CE and CM methods to ensure the best database engine compatibility and fair comparison, while better CE/CM may further improve the performance of Yannakakis$^+$. We first collect basic statistical information from the base tables, including their size, the number of distinct values, the quantiles, etc. Then, during query optimization, we estimate the join size, projection size, and selectivity of selection predicates using some classical methods [34, 36, 45, 56, 61]. Finally, we convert the cardinality estimates into an estimate of the actual running time using a standard cost model.

6 System implementation

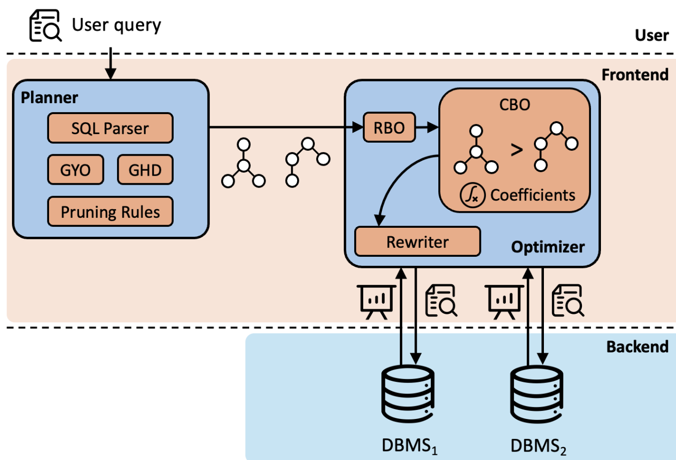


Fig. 8. System Architecture

We have developed a prototype system implementing our algorithms, which consists of two main components: the planner and the optimizer. Figure 8 illustrates the architecture of our system. The planner accepts SQL queries and the database schema via REST APIs. Each input query first undergoes syntax validation using a built-in SQL parser based on Apache Calcite [17]. After validation, the query is transformed into a tree of relational operators. The planner then applies optimizations where appropriate, such as cycle elimination. Next, it builds the hypergraph and generates candidate join trees using the techniques described in Section 5.2. For free-connex queries, each candidate join tree is associated with a subtree $\mathcal{T}_n$, representing the connex subset.

Upon receiving the candidate join trees, the optimizer uses a built-in cost model, along with statistics from the DBMS, to select the optimal join tree. In practice, the planning and optimization steps can be completed within 100 milliseconds. For complex queries that require longer optimization times, our system can choose to skip planning and optimization steps and directly use the join tree provided by the DBMS for the subsequent rewrite step, thereby balancing optimization time and query execution time.

For the rewrite step, our system employs the algorithm described in Section 3 to generate a series of equivalent intermediate representations (IRs) as instructions. Depending on the target DBMSs, the instructions are further converted into executable SQL queries. This design decouples our system from the underlying DBMSs, improving its portability. To support a new target DBMS, only the conversion from rewritten instructions to SQL statements is required. This also allows us to leverage some features of a specific DBMS for tailored performance optimization, enhancing runtime performance. For example, in DuckDB, we utilize temporary views to store the intermediate results of our plan, a method that allows it to follow our algorithm while introducing minimal overhead. Each of the generated SQL statements in our plan is atomic and cannot be further optimized or rewritten. We also verified in the experiments that all these systems executed the given plans as instructed.

Our prototype is available at [7], currently supporting DuckDB [1], PostgreSQL [3], DBMS X (a commercial column-oriented database optimized for analytical processing), and SparkSQL [5].

7 Experimental Evaluation & Analysis

7.1 Experimental Setup

Experimental Environment. Experiments for DuckDB and PostgreSQL were conducted on a machine with an Intel Xeon Gold 6354 CPU @ 3.00GHz (36 cores, 72 threads), 1TB RAM, running Ubuntu 20.04. The software versions used were DuckDB 1.0 and PostgreSQL 16.2. Spark experiments were performed on a machine with an Intel Xeon Silver 4116 CPU @ 2.10GHz (24 cores, 48 threads), 192GB RAM, running AlmaLinux 9.4, using Spark 3.5.1 with Java 1.8.0.

Each query was executed 10 times on each database engine, and we reported the median running time, including both optimization and execution time. I/O time is excluded from the running time. Before running a query, we warm up the database and read all required relations into the memory. A two-hour time limit was set for the SGPB and LSQB benchmarks, and a 30-minute limit for the TPC-H and JOB benchmarks. All systems were used with default configurations, utilizing all available resources: 72 threads for DuckDB and PostgreSQL, and 48 threads for SparkSQL.

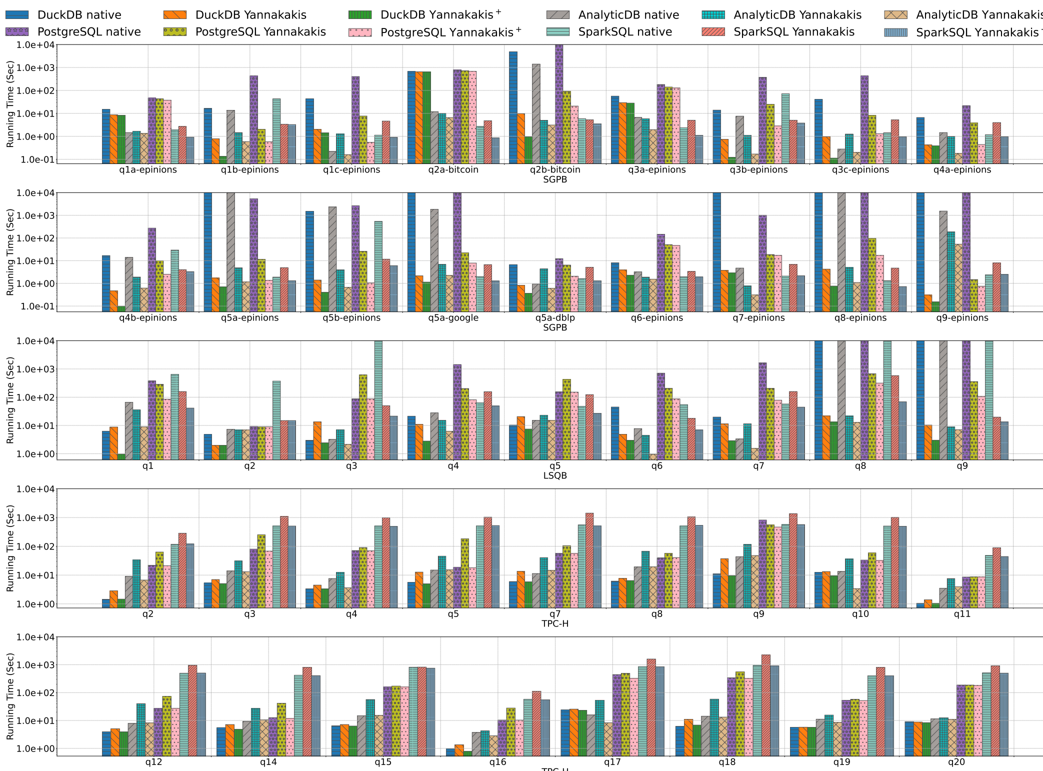
Datasets, Queries, and Benchmarks. We assessed our algorithms using a variety of benchmarks covering graphs, social networks, and relational data to ensure a comprehensive evaluation across different data complexities and join types. All SQL queries used in our evaluation are available in our repository [7].

- **Sub-Graph Pattern Benchmark (SGPB).** We designed queries over diverse graph datasets from the Stanford Network Analysis Project (SNAP) [4], including *bitcoin*, *epinions*, *dblp*, *google*, and *wiki*, containing 24K to 28M edges. These datasets provide a robust test for graph query performance.
- **LSQB.** The LSQB Benchmark [49], derived from the LDBC Social Network Benchmark (LDBC-SNB) [13], focuses on complex queries involving numerous joins typical in social network analysis. We evaluated all nine queries using a scale factor of 30.
- **TPC-H.** TPC-H [6] is an industry-standard benchmark simulating decision support systems with large data volumes and complex queries addressing critical business questions. We conducted experiments using a scale factor of 100.
- **JOB.** The Join Order Benchmark (JOB) [45] comprises 113 analytical queries over the Internet Movie Database (IMDB) dataset [2]. To illustrate performance improvements, we scaled the dataset by enlarging each table 10 to 100 times its original size.
- **CEB.** The Cardinality Estimation Benchmark (CEB) [52, 53] is a benchmark consisting of millions of SQL queries, designed to test the performance of query optimization. It primarily features two workloads: IMDB and StackExchange. Similarly, we have scaled the dataset to 10 times its original size.

For all benchmarks, we focused on evaluating conjunctive queries with aggregations. We omitted operations like `LIMIT` or `ORDER BY` and replaced anti-joins or outer joins with inner joins to standardize the query patterns.

7.2 Results

7.2.1 ***Running Time Comparison***. Figures 9 present the running times and relative speedups of our query rewriter across four benchmarks—SGPB, LSQB, TPC-H, and JOB—evaluated on DuckDB, AnalyticDB, PostgreSQL, and SparkSQL. All bars reaching the axis boundary indicate that the system either exceeded the time limit or encountered memory issues. All the raw experimental results available in our code repository [7].



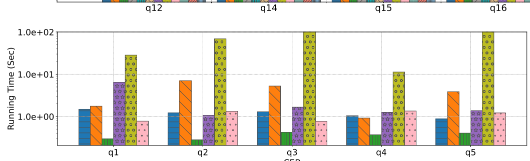


Fig. 9. Running times of DuckDB, AnalyticDB, PostgreSQL, SparkSQL

We first observe that, for the Yannakakis query plans, although they can significantly improve performance by orders of magnitude on queries like SGPB-q4b (35.83x) or SGPB-q5b (1071.78x), they yield significant performance drawbacks on plenty of queries. Especially for queries with PK-FK joins, when executing the Yannakakis plan on the JOB, most queries run slower than their native query plans. Such performance matches the previous observations [29, 54], and the limited improvements are due to (1) The overhead introduced by splitting queries into multiple SQL statements and creating temporary views offsets potential gains. (2) Primary key–foreign key (PK-FK) constraints resulting in intermediate result sizes of $O(N)$, matching the time complexity of the Yannakakis algorithm and leaving little room for optimization.

On the other hand, we observe significant performance improvement in our Yannakakis$^+$ plan. In the total 162 test queries across all platforms/benchmarks, we can achieve performance improvement over 160 queries compared with the native query plans, with an average of 2.4x and a maximum of 47,059x improvement. The performance drawbacks are limited, with 12.75% additional running time at most on the test queries. In addition, we achieved performance improvement over all queries compared with the Yannakakis query plans, with an average of 2.74x and a maximum of 156.03x improvement. The detailed results are:

- **Sub-Graph Pattern Benchmark (SGPB).** Our rewriter significantly enhances performance across all systems on the SGPB benchmark. In DuckDB, we achieve a maximum speedup of

Fig. 10. Speedup achieved by different DBMS on JOB Benchmark.

47,059x and an average of 194x over the native plans. AnalyticDB shows a maximum speedup of 6,606x with an average of 29x, PostgreSQL reaches up to 9,600x with an average of 107x, and SparkSQL records a maximum of 89x and an average of 2.7x. These results highlight the effectiveness of our rewrite algorithm, especially in analytical processing systems like DuckDB.

- **LSQB (Scale Factor 30).** The rewriter provides substantial speedups on the LSQB benchmark. DuckDB experiences a maximum speedup of 2,391x and an average of 14x. AnalyticDB achieves up to 1,016x with an average of 9x, PostgreSQL sees a maximum of 67x and an average of 7x, while SparkSQL records a maximum of 538x with an average of 18x. Notably, several native query plans exceeded time limits or failed due to memory constraints; post-optimization, these queries were completed successfully, particularly Q8 and Q9.
- **TPC-H (Scale Factor 100).** Although the PK-FK constraints on the benchmark also limit the improvement of the new rewriting approach, it is still able to achieve some performance gains and avoid running time drawbacks by optimizing the number of rewritten queries and the query plan for PK-FK joins. DuckDB shows a maximum speedup of 1.33x with an average of 1.06x. AnalyticDB reaches up to 3.93x with an average of 1.20x, PostgreSQL has a maximum of 1.75x and an average of 1.08x, and SparkSQL records a maximum of 1.09x with an average of 1.02x. In addition, our rewrite query plan has at most 12.75% performance drawbacks.
- **JOB.** The rewriter's performance on the Join Order Benchmark (JOB) is mixed. DuckDB achieves a maximum speedup of 14.84x with an average of 1.42x. AnalyticDB reaches up to 94.50x and averages 2.71x, PostgreSQL shows a maximum of 12.31x with an average of 1.40x, and SparkSQL records a maximum of 2.30x and an average of 1.11x. In addition, to provide deeper insights, Table 2 presents statistical analyses of the running times for all 113 queries in the JOB benchmark, which indicate significant improvements across various statistical measures.

- **CEB.** Given the vast number of queries, we selected 5 queries for testing. Notably, the experimental results for DuckDB reach a maximum 5.03x speedup and an average speedup of 3.33x. For PostgreSQL, the maximum speedup was 2.65x, and the average speedup was 1.54x.

Table 2. JOB Statistics

Method (s)	Max	Mean	Med.	Std.Dev.
DuckDB native	933.73	53.02	40.72	113.47
DuckDB Yannakakis	262.67	45.08	44.42	33.17
DuckDB Yannakakis$^+$	67.48	30.12	28.32	20.50
AnalyticDB native	1282.22	106.13	59.78	194.39
AnalyticDB Yannakakis	1468.3	226.19	175.51	220.85
AnalyticDB Yannakakis$^+$	110.28	31.66	19.01	29.72
PostgreSQL native	1289.29	82.66	56.36	147.97
PostgreSQL Yannakakis	422.49	113.52	92.81	89.19
PostgreSQL Yannakakis$^+$	144.16	50.55	43.56	35.68
SparkSQL native	539.37	268.37	201.71	159.64
SparkSQL Yannakakis	1145.17	544.72	430.47	328.92
SparkSQL Yannakakis$^+$	521.33	207.56	170.81	156.95

Table 3. Rule-based Optimization: PK-FK & Annot

JOB-1a (s)	Base	Primitive	PK-FK	Annot	PK-FK & Annot
DuckDB	4.36	29.68	4.51	27.97	3.59
PostgreSQL	7.55	29.18	9.56	14.60	6.95
JOB-4a (s)	**Base**	**Primitive**	**PK-FK**	**Annot**	**PK-FK & Annot**
DuckDB	12.76	32.31	4.28	31.25	4.08
PostgreSQL	10.87	29.11	7.13	28.18	6.72

Table 4. Running Times Under Different Cardinality Estimation Scenarios

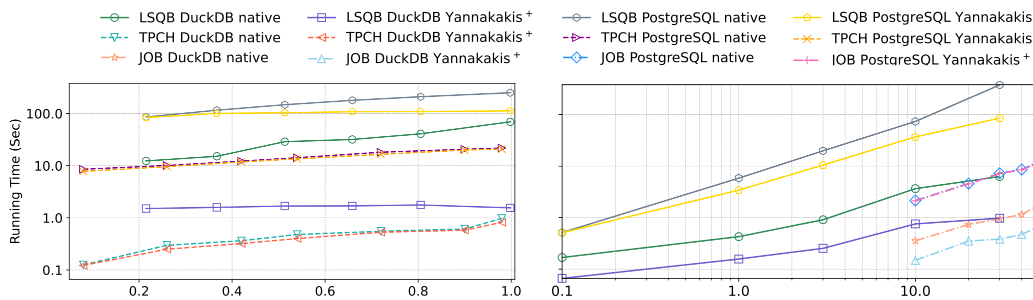
JOB-2b (s)	native	accurate	estimated	worst-case bounds
DuckDB	5.14	4.28	5.10	22.13
PostgreSQL	28.27	10.70	12.82	16.75
JOB-8b (s)	**native**	**accurate**	**estimated**	**worst-case bounds**
DuckDB	23.60	22.74	23.38	38.00
PostgreSQL	92.19	59.86	85.97	97.32
JOB-11d (s)	**native**	**accurate**	**estimated**	**worst-case bounds**
DuckDB	58.58	5.42	7.77	228.21
PostgreSQL	20.06	7.26	10.91	50.10
JOB-17c (s)	**native**	**accurate**	**estimated**	**worst-case bounds**
DuckDB	39.20	16.24	20.46	35.90
PostgreSQL	72.45	69.73	70.30	377.29
JOB-27b (s)	**native**	**accurate**	**estimated**	**worst-case bounds**
DuckDB	41.49	40.46	41.40	53.81
PostgreSQL	38.85	21.72	38.30	79.3

7.2.2 ***Effectiveness of the Rule-based Optimization.*** We conducted ablation experiments to test the effects of two rules: PK-FK projection elimination and pruning for annotation. We select 1a and 4a query from the JOB benchmark, where *base* represents the effect without any rewrite, *primitive* represents the result without both rewrite rules, *PK-FK* represents the effect with only projection elimination, *Annot* represents the effect with only pruning for annotation, and *PK-FK & Annot* represents the combined effect of both optimizations. We test the experimental performance under two DBMSs and find that applying both optimizations simultaneously yields excellent experimental results, as shown in Table 3.

7.2.3 ***Effectiveness of Cardinality Estimation.*** To test our cost-based optimizer, we evaluate the impact of cardinality estimation accuracy on query performance under three scenarios:

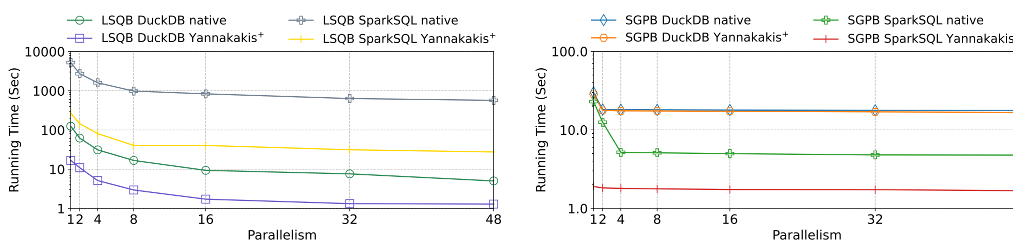
- **Accurate Cardinality**: The optimizer uses exact sizes for all intermediate query results.
- **Estimated Cardinality**: The optimizer relies on estimates based on available statistics like cardinalities and the number of distinct values (NDV).
- **Worst-Case Bounds**: The optimizer assumes maximum possible join sizes (Cartesian product) unless key constraints are present.

Table 4 presents the execution times for three queries on DuckDB and PostgreSQL under these scenarios, along with the native plans. The results indicate that accurate cardinality leads to optimal performance, while with estimated statistics, execution times improve significantly over the native plans and can provide similar performance compared with the optimal estimation. On the other hand, we also need some accuracy to ensure the performance, as if we only apply the worst-case estimation, the performance can be much worse than our current selection or even native plans.



(a) Selectivity (b) Scale

Fig. 11. Running times of different selectivity & scale.



(a) LSQB-Q1 (b) SGPB-Q1

Fig. 12. Running times under different parallelism.

Table 5. Optimizaiton time for different queries.

Query (s)	DuckDB native	DuckDB Yannakakis$^+$	PostgreSQL native	PostgreSQL Yannakakis$^+$	#Tables	#Attributes	Opt-Time	DuckDB Opt-Time
SNAP-q1a	15.10	8.19	46.96	37.64	3	6	0.133703232	0.0021
SNAP-q6	8.12	2.29	146.01	46.69	5	6	0.236415863	0.0014
LSQB-q1	6.27	0.97	376.31	85.34	10	7	0.065845966	0.0087
LSQB-q5	10.37	7.47	153.74	151.53	3	4	0.087508917	0.0017
TPCH-q3	5.32	5.07	79.99	68.39	3	8	0.071694851	0.0019
TPCH-q10	12.36	9.32	33.25	32.24	4	13	0.085761070	0.0027
TPCH-q19	5.72	5.68	52.94	51.89	2	9	0.074432135	0.0020
JOB-1a	3.66	3.21	7.34	6.72	5	8	0.075666189	0.0027
JOB-10c	23.59	23.49	131.33	92.96	7	10	0.172396183	0.0051
JOB-21a	40.93	40.01	56.36	36.78	9	13	0.080693007	0.0137
JOB-27c	41.10	40.76	51.22	36.65	12	17	0.086112976	0.0594
JOB-30a	61.14	35.86	60.24	49.86	14	21	0.096741199	0.0666

7.2.4 ***Robustness.*** From the experimental results, Yannakakis$^+$ consistently shows improvements in the vast majority of queries tested and shows excellent robustness. This is mainly benefiting from Yannakakis$^+$'s optimization of the number of semi-joins. In most queries, only one round or even no semi-join reduction is required. The rule-based optimizer also helps avoid unnecessary semi-join reductions. Beyond semi-join reduction, aggregation pushdown also contributes to improved performance. Furthermore, we also tested Yannakakis$^+$ under different selectivity and data scales to further illustrate its robustness along these dimensions:

Selectivity. We selected two queries and altered the predicate to change F (the full join size). In Figure 11(a), the horizontal axis values represent the percentage of output value (size) compared with output value (size) without a predicate. It can be observed that as the output size increases, the advantages of the rewriter increase compared to the original query execution.

Scale. We selected five settings for LSQB: 0.1, 1, 3, 10, and 30. For JOB, we chose five scales with equal intervals from 10 to 50. The curves in Figure 11(b) show that as the scale increases, the running time generally increases proportionally, and the larger the scale, the better the execution performance of our rewrite.

Parallel Query Processing. Our approach is pure relational, which offers the significant benefit of seamless adoption in other computing scenarios, such as parallel processing. We conducted an experiment where we varied the number of threads utilized by each DBMS and re-executed a set of specific queries, with LSQB-Q1 selected in Figure 12(a) and SGPB-Q1 in Figure 12(b). From the experimental results, our new query plan also shows improvement with additional threads, which is similar to that of the native query plan, indicating great parallelization of our new plan.

Optimization Time. Finally, it is worth mentioning that the optimization overhead is relatively small compared to the total query execution time. We selected representative 12 queries from the four benchmarks to investigate the relationship between optimization time and the number of tables and attributes within the query. As seen from Table 5, the optimization time is mostly kept within 100ms, which is negligible compared to the query execution time. This favorable outcome is partially attributed to the introduction of a hint mechanism within our system. When the optimization time reaches a certain threshold, we leverage the existing plans from the DBMS to assist our estimator. We do observe that there is a performance gap between the optimization time of Yannakakis$^+$ and that of DuckDB. However, this gap is well offset by the performance gains in the query execution time. Furthermore, we believe this gap can be significantly reduced if we integrate Yannakakis$^+$ within the database kernel—recall that the current implementation of our optimizer is outside the engine, thus incurring quite some overhead (in exchange for better compatibility with different engines).

8 Conclusion and Future Work

In this work, we introduce Yannakakis$^+$, an improved version of the original Yannakakis algorithm. This new version not only maintains the theoretical guarantees but is also highly efficient in practice. The experimental results suggest that Yannakakis$^+$ can not only achieve order-of-latitude improvements on specific queries while avoiding regressions on other queries.

Our current implementation of Yannakakis$^+$ follows a rewrite-based approach to showcase its applicability across a wide range of database and data processing systems (row-based vs column-based, centralized vs distributed). The next natural step is to integrate it into an SQL engine, which could further improve its performance. For example, we can generate a single physical plan instead of issuing multiple SQL statements, reduce the communication overhead between system components, and eliminate repeated parsing, plan generation, and optimization. Beyond these direct advantages, combining Yannakakis$^+$ with the database engine offers further opportunities for optimization: (1) Note that our use of semijoin is "soft", i.e., it is alright to leave a small number of dangling tuples unremoved. So this can be using Bloom filters, which are much more efficient than using the existing semi-join operator. (2) Implementing Yannakakis$^+$ inside database engines enables access to more sophisticated database statistics, potentially improving the cost-based optimizer with tailored-made CE/CM. (3) The bottleneck of the optimization time in Yannakakis$^+$ is running GYO and GHD to enumerate all possible query plans. By integrating Yannakakis$^+$ within database engines, it is possible to reduce redundant plan enumeration through the native database optimizer, which is especially useful for large queries with hundreds of relations or attributes. These enhancements are expected to boost Yannakakis$^+$'s performance and make it a more robust solution for advanced query processing scenarios.

References

[1] DuckDB. https://duckdb.org/.

[2] IMDB. http://www.imdb.com/.
[3] PostgreSQL. https://www.postgre.org/.
[4] SNAP. https://snap.stanford.edu/snap/.
[5] SparkSQL. https://spark.apache.org/sql/.
[6] TPC-H. https://www.tpc.org/tpch/.
[7] Yannakakis$^+$: Practical Acyclic Query Evaluation with Theoretical Guarantees, Code Repository. https://github.com/hkustDB/Yannakakis-Plus.
[8] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246
[9] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Addison-Wesley Longman Publishing Co., Inc.
[10] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16).* Association for Computing Machinery, New York, NY, USA, 13–28. https://doi.org/10.1145/2902251.2902280
[11] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-Type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17).* Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3034786.3056105
[12] Jayadev Acharya, Ilias Diakonikolas, Chinmay Hegde, Jerry Zheng Li, and Ludwig Schmidt. 2015. Fast and Near-Optimal Algorithms for Approximating Distributions by Histograms. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Melbourne, Victoria, Australia) *(PODS '15).* Association for Computing Machinery, New York, NY, USA, 249–263. https://doi.org/10.1145/2745754.2745772
[13] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299
[14] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science.* 739–748. https://doi.org/10.1109/FOCS.2008.43
[15] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic.* Springer Berlin Heidelberg, Berlin, Heidelberg, 208–222.
[16] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. 1983. On the desirability of acyclic database schemes. *JACM* 30, 3 (1983), 479–513.
[17] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18).* Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662
[18] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2024. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. *arXiv preprint arXiv:2411.04042* (2024).
[19] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3215–3228. https://doi.org/10.14778/3681954.3681995
[20] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19).* Association for Computing Machinery, New York, NY, USA, 18–35. https://doi.org/10.1145/3299869.3319894
[21] Nofar Carmeli and Markus Kröll. 2021. On the Enumeration Complexity of Unions of Conjunctive Queries. *ACM Trans. Database Syst.* 46, 2, Article 5 (may 2021), 41 pages. https://doi.org/10.1145/3450263
[22] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. San Francisco, 146–155.
[23] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17).* Association for Computing Machinery, New York, NY, USA, 759–774. https://doi.org/10.1145/3035918.3035921
[24] David DeHaan and Frank Wm. Tompa. 2007. Optimal top-down join enumeration. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07).* Association for Computing Machinery, New York, NY, USA, 785–796. https://doi.org/10.1145/1247480.1247567
[25] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference*

on Management of Data. 1241–1258.

[26] R. Fagin. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM* 30, 3 (1983), 514–550.

[27] Pit Fender and Guido Moerkotte. 2013. Counter strike: generic top-down join enumeration for hypergraphs. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1822–1833. https://doi.org/10.14778/2556549.2556565

[28] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

[29] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice. *arXiv preprint arXiv:2303.02723* (2023).

[30] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 1999. Hypertree Decompositions and Tractable Queries. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (, Philadelphia, Pennsylvania, USA,) *(PODS '99)*. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/303976.303979

[31] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM* 56, 6, Article 30 (sep 2009), 32 pages. https://doi.org/10.1145/1568318.1568320

[32] MH Graham. 1980. *On the universal relation.* University of Toronto. Computer Systems Research Group.

[33] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1, Article 4 (aug 2014), 20 pages. https://doi.org/10.1145/2636918

[34] Dimitrios Gunopulos, George Kollios, J. Tsotras, and Carlotta Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. 14, 2 (April 2005), 137–154. https://doi.org/10.1007/s00778-003-0090-4

[35] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[36] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. 1995. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 311–322.

[37] Stockinger K Heitz J. 2019. Join query optimization with deep reinforcement learning algorithms. *CoRR* abs/1911.11689 (2019). arXiv:1911.11689 [cs.DB]

[38] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*. 857–872.

[39] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (March 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[40] Xiao Hu and Qichen Wang. 2023. Computing the Difference of Conjunctive Queries Efficiently. *Proc. ACM Manag. Data* 1, 2, Article 153 (jun 2023), 26 pages. https://doi.org/10.1145/3589298

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

[42] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293

[43] Raghav Kaushik and Dan Suciu. 2009. Consistent histograms in the presence of distinct value counts. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 850–861. https://doi.org/10.14778/1687627.1687723

[44] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment* 12, 5 (2019), 502–515.

[45] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[46] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for SQL queries using statistical techniques. *Proc. VLDB Endow.* 5, 11 (July 2012), 1555–1566. https://doi.org/10.14778/2350229.2350269

[47] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[48] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[49] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Virtual Event, China) *(GRADES-NDA '21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3461837.3464516

[50] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 930–941.

[51] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 539–552. https://doi.org/10.1145/1376616.1376672

[52] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (July 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[53] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[54] Thomas Neumann. 2024. Closing the Gap between Theory and Practice in Query Optimization. In *Companion of the 43rd Symposium on Principles of Database Systems* (Santiago AA, Chile) *(PODS '24)*. Association for Computing Machinery, New York, NY, USA, 4. https://doi.org/10.1145/3635138.3654765

[55] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (mar 2018), 40 pages. https://doi.org/10.1145/3180143

[56] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record* 25, 2 (1996), 294–305.

[57] RelationalAI. Accessed: 01.10.2024. Worst-case Optimal Join Algorithms. https://relational.ai/resources/worst-case-optimal-join-algorithms

[58] Florin Rusu and Alin Dobra. 2008. Sketches for size of join estimation. *ACM Trans. Database Syst.* 33, 3, Article 15 (Sept. 2008), 46 pages. https://doi.org/10.1145/1386118.1386121

[59] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[60] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3318464.3380584

[61] Arun Swami and K. Bernhard Schiefer. 1994. On the estimation of join result sizes. In *Proceedings of the 4th International Conference on Extending Database Technology: Advances in Database Technology* (Cambridge, United Kingdom) *(EDBT '94)*. Springer-Verlag, Berlin, Heidelberg, 287–300.

[62] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2013. Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal* 22, 1 (Feb. 2013), 3–27. https://doi.org/10.1007/s00778-012-0293-7

[63] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P. Chakkappen. 2015. Join size estimation subject to filter conditions. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1530–1541. https://doi.org/10.14778/2824032.2824051

[64] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *arXiv preprint* (2025).

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

[66] Qichen Wang, Qiyao Luo, and Yilei Wang. 2024. Relational Algorithms for Top-k Query Evaluation. *Proc. ACM Manag. Data* 2, 3, Article 168 (may 2024), 27 pages. https://doi.org/10.1145/3654971

[67] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3514221.3517830

[68] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1969–1981. https://doi.org/10.1145/3448016.3452808

[69] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proc. VLDB Endow.* 12, 3 (Nov. 2018), 210–222. https://doi.org/10.14778/3291264.3291267

[70] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1721–1736. https://doi.org/10.1145/2882903.2882914

[71] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf

[72] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

[73] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979*. IEEE, 306–312.

[74] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/2463676.2463701

[75] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. In *SIGMOD*.

Received October 2024; revised January 2025; accepted February 2025