



Database Theory in Action: Yannakakis' Algorithm

Paraschos Koutris  

University of Wisconsin-Madison, WI, USA

Stijn Vansummeren  

UHasselt, Data Science Institute, Diepenbeek, Belgium

Qichen Wang  

Nanyang Technological University, Singapore

Yisu Remy Wang  

University of California, Los Angeles, CA, USA

Xiangyao Yu  

University of Wisconsin-Madison, WI, USA

Abstract

Yannakakis' seminal algorithm is optimal for acyclic joins, yet it has not been widely adopted due to its poor performance in practice. This paper briefly surveys recent advancements in making Yannakakis' algorithm more practical, in terms of both efficiency and ease of implementation, and points out several avenues for future research.

2012 ACM Subject Classification Information systems → Join algorithms

Keywords and phrases Join algorithms, acyclicity, Yannakakis' algorithm

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.25

Funding *Stijn Vansummeren*: supported by Hasselt University Bijzonder Onderzoeksfonds (BOF) under Grant BOF20ZAP02 and the Research Foundation Flanders (FWO) under Grant No. G0B9623N. *Qichen Wang*: supported by the Nanyang Technological University Startup Grant and the Singapore Ministry of Education under Grant No. RS32/25.

1 Introduction

In 1981, an optimal algorithm for computing acyclic joins was concurrently discovered by Bernstein et al. [3, 4] and Yannakakis [16], and has since become known as Yannakakis' algorithm (hereafter YA). The algorithm is conceptually simple, applicable to a wide class of practical queries, and has the strong guarantee of being *instance optimal*, which means it has the best possible asymptotic complexity for *any* input instance. Concretely, recall that a natural join query $Q = R_1(\mathbf{x}_1) \bowtie \cdots \bowtie R_n(\mathbf{x}_n)$ is α -*acyclic* if there exists a *join tree*, which is a tree whose nodes are the relations of Q , and for each variable x in Q , the set of nodes $\{R_i \mid x \in \mathbf{x}_i\}$ is connected. To evaluate an α -acyclic query Q , YA makes two passes over the query's join tree, using semijoins to remove dangling tuples; then it makes a final pass to produce the output. For example, given $Q = R(i, j) \bowtie S(j, k) \bowtie T(k, l) \bowtie U(l, m)$ with join tree $R \rightarrow S \rightarrow T \rightarrow U$ where R is the root, YA computes the following relations in sequence: $T' = T \bowtie U$, $S' = S \bowtie T'$, $R^* = R \bowtie T'$ (bottom-up pass); $S^* = S' \bowtie R^*$, $T^* = T' \bowtie S^*$, $U^* = U \bowtie T^*$ (top-down pass); and $Q = R^* \bowtie S^* \bowtie T^*$ (final output).

Despite YA's simplicity and instance-optimality, none of today's mainstream database systems implement the algorithm, mainly due to the high constant factor overhead involved in performing the semijoin reduction phases. For instance, recent experiments by Gottlob et al. [8] demonstrate that while YA improves tail-end performance for long-running queries, it increases run time by an average of 2.4× compared to standard binary joins. Furthermore, the algorithm's three-pass design complicates its integration into existing query optimizers.



© Paraschos Koutris, Stijn Vansummeren, Qichen Wang, Yisu Remy Wang, and Xiangyao Yu; licensed under Creative Commons License CC-BY 4.0

29th International Conference on Database Theory (ICDT 2026).

Editors: Balder ten Cate and Maurice Funk; Article No. 25; pp. 25:1–25:6



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In response to these observations, recent research has proposed several approaches to make YA more practical: more efficient semijoin operator implementations; reducing the number of semijoin passes; and creating novel join operators that can achieve the theoretical benefits of YA without additional overhead. In this paper we provide a brief survey of these approaches, distill the key insights behind them, and identify open challenges for future research.

2 Improving Semijoin Efficiency with Bitvector Filters

Bloom filter [6] is a space-efficient, probabilistic data structure for sets. By mapping each element in a set to k bit positions in a vector using k independent hash functions, it supports set membership queries with a configurable one-sided error: false positives are possible but there will be no false negatives. This property makes Bloom filters very effective for pre-filtering join relations while lowering memory usage, and they are more cache-friendly compared to hash tables. To perform a pre-filtering pass for a binary join $R \bowtie S$, a system can first build a Bloom filter on the join keys from one relation (e.g., S) and then probe this filter with the join keys from the other relation (R). Tuples in R whose keys fail the membership test are guaranteed not to join with S and can be safely discarded. While false positives may allow some non-joining tuples to pass through, these are subsequently eliminated during the final join on the pre-filtered relations.

Integrating bitvector filters such as Bloom filters directly into a cost-based optimizer is non-trivial, as they can cause the plan space to grow exponentially with the number of relations. However, for the common cases of star and snowflake schemas with primary-key-foreign-key joins, it has been shown that the optimal plan within the space of right-deep trees can be found by evaluating only a linear number of candidate plans, which are robust against the join ordering of dimension tables [7]. Further research has focused on integrating the selection of filter types [11] and their configuration into the optimizer's cost model [17].

Recently, *predicate transfer* (PT) [15] was proposed to approximate semijoin using Bloom filters. By decoupling the pre-filtering phase from the join phase, PT improves the robustness of the subsequent join ordering decision. Using a simple heuristic of transferring predicates from smaller relations to larger ones, PT often achieves similar performance across different join orders. Building on this, *Robust Predicate Transfer* (RPT) [18] refines PT to restore the theoretical guarantees of YA for acyclic queries and implement it into DuckDB's query engine. Instead of the simple small-to-large assumption, RPT chooses a "root" relation to maximize the downstream reduction. Furthermore, RPT requests the subsequent join order to be *monotone*. When the query is γ -acyclic, any join order is monotone, while for general α -acyclic queries, such monotone orders will correspond to a specific join tree.

Several research challenges remain. Integrating bitvector filters into a cost-based optimizer is essential, as the overhead of the pre-filtering phase may not be justified for unselective queries. Further investigation is needed to better understand how to leverage existing indexes, such B-trees. Finally, joins and filters often come with downstream operations like aggregations. When the bitvector filter needs to carry payloads, one can no longer benefit from the Bloom filter, as the false positives can violate the final correctness of the query.

3 Unlocking Theoretical Efficiency with Query Rewriting

YA is a pure *relational* algorithm [14]; its logic is data independent and can be expressed as a composition of entirely standard relational operators. This property makes it possible to implement YA-style evaluation via query rewriting [8, 13, 12], which separates planning from

execution. An external planner first constructs a structure-aware query plan based on YA, which is then translated into SQL for execution by an unmodified DBMS. This translation can take the form of a script of sequential SQL statements that materialize intermediate subqueries [8], or a single, complex DAG plan that allows the underlying engine to perform more holistic optimizations such as operator pipelining [13].

Nevertheless, a rewrite-based implementation of YA can incur performance overhead, particularly on queries with simple PK-FK joins where traditional optimizers are highly effective [8, 13]. Recent work has focused on algorithmic refinements of YA itself to minimize the number of semijoins required. In fact, the three-phase YA can collapse into two phases by eliminating the top-down semijoins, when the join order respects the hierarchical structure of the join tree [1]. Building on this insight, YA⁺ [13] formalizes a cost-based optimization for YA. The high-level idea is to maximize the filtering power of the single semijoin round by carefully choosing the join tree and the semijoin order. For an acyclic query, YA⁺ enumerates all join trees for the query and estimates the size of intermediate results after a bottom-up semijoin pass for each tree. It then selects the join tree and corresponding semijoin scheduling that minimizes the intermediate result sizes. This strategy yields much more robust performance compared to conventional binary join plans: it incurs virtually no slowdown in the worst case, yet it can still achieve order-of-magnitude speedups on long-running queries.

Extending YA-style evaluation to other relational operators like aggregation is also realized through query rewriting. For example, aggregation can be supported by evaluating YA over annotated relations, where each tuple carries annotations used to compute aggregates [10]. This abstraction allows aggregation to be pushed down. YA⁺ [13] integrates this idea with its two-phase strategy, maintaining annotations during the semijoins. Furthermore, when a query’s group-by attributes are entirely contained in a single relation, the query is said to be relation-dominated [13] or OMA [12]. In such cases, the final join phase of YA can be omitted, and aggregation can be performed directly on the filtered relation after semijoin pruning.

Query rewriting offers a lightweight and portable path to integrate YA-style evaluation into existing database systems without modifying their core components. It is also orthogonal to physical-level optimizations, such as using Bloom filters. However, this approach introduces certain overheads in practice. Because the logical plan is fixed externally, the rewritten queries may bypass some of the system’s built-in optimization rules, potentially leading to suboptimal plans. The workflow also incurs extra planning time and communication costs during optimization. While the rewriting approach is effective for proof-of-concept deployments, it motivates the need for native support of YA-style strategies within the optimizer and execution engine. A direct implementation could preserve the structural advantages of YA while minimizing the operational overhead of external rewrites.

4 Achieving Optimality without Regression

The techniques discussed so far *reduce* the overhead of YA in different ways. In this section, we turn to a class of recent algorithms that have provable guarantees to incur *zero overhead*. An algorithm *A* is said to be zero-overhead with respect to a baseline algorithm *B* if the former has lower cost (according to some cost model) for every query plan. Specifically, the goal is to design algorithms that are zero-overhead with respect to the binary hash join.

Traditional hash join consists of two phases: the first phase builds hash tables for certain relations, and the second probes into these hash tables to find matching tuples. The intuition behind zero-overhead algorithms is to “piggyback” semijoin reduction into one of these phases, thereby achieving instance-optimality “for free”. The algorithms fall into two classes:

bottom-up algorithms perform semijoin reduction during the build phase, while top-down algorithms do so during the probe phase. These classes are dual to each other: while bottom-up algorithms guarantee zero overhead for right-deep query plans, top-down algorithms assume left-deep plans. We adopt the convention to build hash tables on the right side of each binary join operator. Bottom-up algorithms are therefore most effective when hash table construction accounts for the main cost during execution, whereas top-down algorithms reduce the cost of probing hash tables and perform well when indexes are available.

The first bottom-up algorithm was proposed by Birler, Kemper, and Neumann [5] for compiled query engines, and later formalized using nested relational algebra by Bekkers et al. [2] who generalized it to support other relational operators and adapted it for vectorized, column-oriented systems. The key idea of these algorithms is to introduce a new *nested semijoin* operator \triangleright that is “in between” a join and a semijoin. During the execution of $R \triangleright S$, the operator scans R and probes into the hash table for S . For each tuple $r \in R$, instead of materializing all matching tuples $s \in S$, the operator attaches to r a pointer to these matches, thereby *nesting* the matches under r . The algorithms then build up a nested representation of the join result by computing the nested semijoins bottom-up along the join tree, and finally unnest the representation to produce the output. Throughout this process, each “nested join” runs in time linear in the size of each input relation, and the final unnesting step takes linear time in the size of the output. Bekkers et al. proved the algorithm to be zero-overhead for a class of plans that generalizes right-deep plans.

Hu, Wang, and Miranker [9] proposed a top-down algorithm called TreeTracker Join. They present the algorithm as a simple modification of the traditional (pipelined) binary hash join: whenever a hash lookup fails, the algorithm backtracks to the tuple responsible for the failure and removes it from its relation. To demonstrate the main ideas of the algorithm, let us consider the same query Q in Section 1. Given the left-deep query plan $(R \bowtie (S \bowtie (T \bowtie U)))$, and assume hash tables are built for S , T , and U , the algorithm first iterates over R , probing into the hash table for S . Suppose for the tuple $(i_1, j_1) \in R$ this results in a match $(j_1, k_1) \in S$ which the algorithm uses to probe into T . This again returns a match $(k_1, l_1) \in T$, but probing into U with that tuple fails to return any match. Because the join key l_1 was introduced by the T relation, the algorithm proceeds to delete (k_1, l_1) from T , thereby removing that dangling tuple “on the fly”. Hu, Wang, and Miranker proved the algorithm to be instance-optimal and zero-overhead for left-deep plans.

Compared to other approaches, implementing the new operators in zero-overhead algorithms requires greater effort. On the other hand, these algorithms offer real performance gains, and their strong guarantee of regression avoidance is critical for practical deployment.

5 Conclusion and Open Problems

Each approach to improve YA discussed makes a different tradeoff between ease of implementation and performance, and reduces algorithm overhead in different ways. To fully realize the potential of optimal join algorithms, however, requires overcoming additional challenges. For example, although some of the approaches are compatible with indexes, fully leveraging all available indexes remains an interesting problem. There is also ongoing debate on the role of query optimization in the context of optimal algorithms: while RPT has been shown to perform well regardless of the query plan, a better plan does lead to faster execution in other approaches. Query optimization for acyclic joins also requires different methods of plan enumeration and cost estimation.

References

- 1 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. doi:10.1007/978-3-540-74915-8_18.
- 2 Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, 2025. doi:10.14778/3742728.3742737.
- 3 Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981. doi:10.1145/322234.322238.
- 4 Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981. doi:10.1145/319628.319650.
- 5 Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proc. VLDB Endow.*, 17(11):3215–3228, 2024. doi:10.14778/3681954.3681995.
- 6 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. doi:10.1145/362686.362692.
- 7 Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. Bitvector-aware query optimization for decision support queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, pages 2011–2026, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389769.
- 8 Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice. *arXiv preprint arXiv:2303.02723*, 2023. doi:10.48550/arXiv.2303.02723.
- 9 Zeyuan Hu, Yisu Remy Wang, and Daniel P Miranker. Treetracker join: Simple, optimal, fast. *arXiv preprint arXiv:2403.01631*, 2024.
- 10 Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '16*, pages 91–106, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2902251.2902293.
- 11 Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proc. VLDB Endow.*, 12(5):502–515, January 2019. doi:10.14778/3303753.3303757.
- 12 Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. Avoiding materialisation for guarded aggregate queries. *Proc. VLDB Endow.*, 18(5):1398–1411, 2025. doi:10.14778/3718057.3718068.
- 13 Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proc. ACM Manag. Data*, 3(3), June 2025. doi:10.1145/3725423.
- 14 Qichen Wang, Qiyao Luo, and Yilei Wang. Relational algorithms for top-k query evaluation. *Proc. ACM Manag. Data*, 2(3), May 2024. doi:10.1145/3654971.
- 15 Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 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, 2024. URL: <https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf>.
- 16 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

25:6 Database Theory in Action: Yannakakis' Algorithm

- 17 Tim Zeyl, Qi Cheng, Reza Pournaghi, Jason Lam, Weicheng Wang, Calvin Wong, Chong Chen, and Per-Ake Larson. Including bloom filters in bottom-up optimization. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS '25*, pages 703–715, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3722212.3724440.
- 18 Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proc. ACM Manag. Data*, 3(3), June 2025. doi:10.1145/3725283.