

# Reaching Back to Move Forward: Using Old Ideas to Achieve a New Level of Query Optimization

Georg Gottlob<sup>1</sup>, Matthias Lanzinger<sup>1</sup>, Davide Mario Longo<sup>2</sup>, Cem Okulmus<sup>3</sup>,  
Reinhard Pichler<sup>2</sup> and Alexander Selzer<sup>2</sup>

<sup>1</sup>University of Oxford, UK

<sup>2</sup>TU Wien, Austria

<sup>3</sup>Umeå University, Sweden

## Abstract

Join queries involving many relations pose a severe challenge to today's query optimisation techniques. To some extent, this is due to the fact that these techniques do not pay sufficient attention to structural properties of the query. In stark contrast, the Database Theory community has intensively studied structural properties of queries (such as acyclicity and various notions of width) and proposed efficient query evaluation techniques through variants of Yannakakis' algorithm for many years. However, although most queries in practice actually are acyclic or have low width, structure-guided query evaluation techniques based on Yannakakis' algorithm have not found their way into mainstream database technology yet.

The goal of this work is to address this gap between theory and practice. We want to analyse the potential of considering the query structure for speeding up modern DBMSs in cases that have been traditionally challenging. To this end, we propose a rewriting of SQL queries into a sequence of SQL statements that force the DBMS to follow a Yannakakis-style query execution. Through first empirical results we show that structure-guided query evaluation can indeed make the evaluation of many difficult join queries significantly faster.

## Keywords

query optimization, large join queries, Yannakakis' algorithm

## 1. Introduction

Query processing lies at the very heart of database applications and systems – with join queries arguably being the most fundamental and basic form of queries. A lot of research spanning over several decades has gone into optimizing queries in general and join queries in particular. Consequently, in many practical cases, Database Management Systems (DBMSs) perform really well. However, there still remain queries where today's DBMSs struggle or simply fail. This is

---


AMW 2023: 15th Alberto Mendelzon International Workshop on Foundations of Data Management

✉ georg.gottlob@cs.ox.ac.uk (G. Gottlob); matthias.lanzinger@cs.ox.ac.uk (M. Lanzinger);  
davide.longo@tuwien.ac.at (D. M. Longo); okulmus@cs.umu.se (C. Okulmus); reinhard.pichler@tuwien.ac.at  
(R. Pichler); alexander.selzer@tuwien.ac.at (A. Selzer)

🆔 0000-0002-2353-5230 (G. Gottlob); 0000-0002-7601-3727 (M. Lanzinger); 0000-0003-4018-4994 (D. M. Longo);  
0000-0002-7742-0439 (C. Okulmus); 0000-0002-1760-122X (R. Pichler); 0000-0002-6867-5448 (A. Selzer)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

especially the case with queries that involve the join of many (say 10, 50 or even hundreds of) relations. Large join queries remain challenging even when all joins are made along foreign key constraints, one of the most common and basic cases for relational DBMSs. Such queries are becoming more and more common, e.g., when automatically generated by business intelligence tools [1]. It, therefore, is a requirement for DBMSs today to cope also with simply structured queries of this order of magnitude.

The traditional approach to evaluating a join query is to split it into a sequence of two-way joins. One of the main tasks of query optimisation is then to determine an optimal join order, which avoids the costly computation of large intermediate results as far as possible. However, even for moderately large queries, the resulting optimisation problem becomes too difficult to solve exactly and DBMSs resort to heuristic optimisation techniques. For instance, PostgreSQL 14 by default performs a full search for the optimal plan only up to 11 joins. Sophisticated pruning methods and parallelisation have been shown to push this threshold higher [2], but the task still remains challenging. Moreover, the problem of huge intermediate results is an intrinsic deficiency of splitting join queries into a sequence of two-way joins and not restricted to the choice of a bad join order. Worst-case optimal joins [3] provide a solution to this problem for queries of particular structure (small cyclic queries with joins that do not follow foreign key relationships) and heavily skewed data. But empirical studies of database queries have shown that most queries in practice are acyclic or almost acyclic, involving mostly joins along foreign keys [4, 5] and thus call for a different solution.

From a theoretical perspective, the problem of avoiding large intermediate results in join queries has long been considered as mostly solved. For acyclic queries, Yannakakis' algorithm [6] is well known to guarantee query answering without any unnecessary intermediate results by following the inherent tree-like structure of acyclic queries in the evaluation of the query. From there, a rich theory of structural decompositions and related notions of width has been developed [7, 8] that generalises the acyclic case to general queries with guaranteed bounds (relative to some notion of width) on the intermediate results.

A small number of research systems have indeed adopted structural decomposition methods and worst-case optimal join algorithms with highly promising results [9, 10, 11, 12, 13]. However, these are "standalone-systems" (i.e., not integrated into widely used relational DBMSs) and some of the most successful approaches have actually focused on queries that are more of graph theoretical interest and particularly suitable to worst-case optimal joins (e.g., queries with clique, or lollipop graph structure). Therefore, despite all the progress made with applying Yannakakis-style query evaluation to real-world problems and the experience gained with these systems, the principal question remains unanswered:

*Can the theory of Yannakakis-style query evaluation of acyclic queries be of use for typical relational queries in mainstream RDBMSs?*

Our goal is to study precisely this question and to bridge this gap between the theory and systems communities. We thus embark on a broad experimental evaluation on a recent benchmark [2] that is representative of the typical yet challenging queries we are interested in: big, (almost) acyclic, with all joins being along foreign key constraints.

Such an evaluation on mainstream DBMSs is not straightforward due to an apparent mismatch in paradigms between Yannakakis' algorithm, which operates in multiple phases, and

the Volcano iterator model commonly adapted by modern DBMSs. A direct integration of such methods is therefore laborious and shifts the performance question towards a matter of effective implementation and integration, rather than a study of the general viability of the method. Instead, we base our experimental evaluation on a DBMS-agnostic rewriting-based approach to control a Yannakakis-style evaluation from “outside” the DBMS by submitting to the DBMS appropriate SQL statements that correspond to the operations performed by Yannakakis’ algorithm. Using these rewritings, we compare the performance of a structure-guided approach to the standard query execution strategies in three DBMSs: PostgreSQL, DuckDB, and Spark SQL, that were selected as popular representatives of distinct types of DBMS architecture. Our experimental results thus obtained clearly indicate that structure-guided evaluation can indeed bring large performance gains and can thus alleviate some of the most critical pain-points of modern DBMSs.

## 2. Preliminaries

We assume familiarity with basic database terminology and we very briefly recall only the most important concepts. Consider a *Conjunctive Query* (CQ)  $Q$  as Relational Algebra expressions of the form  $Q = \pi_U(R_1 \bowtie \dots \bowtie R_n)$ , where  $R_1, \dots, R_n$  are pairwise distinct relations and the projection list  $U$  consists of attributes occurring in the  $R_i$ ’s. Then  $Q$  is *acyclic*, if it has a *join tree*, i.e., a rooted, labelled tree  $\langle T, r, \lambda \rangle$  with root  $r$ , such that (1)  $\lambda$  is a bijection that assigns to each node of  $T$  one of the relations in  $\{R_1, \dots, R_n\}$  and (2) if some attribute  $A$  occurs in both relations  $\lambda(u_i)$  and  $\lambda(u_j)$  for two nodes  $u_i$  and  $u_j$  in  $T$ , then  $A$  occurs in the relation  $\lambda(u)$  for every node  $u$  along the path between  $u_i$  and  $u_j$ . Deciding whether a CQ is acyclic and constructing a join tree only requires linear time. Yannakakis’ algorithm [6] to efficiently evaluate acyclic CQs proceeds in 3 traversals of the join tree  $T$ : (1) in a bottom-up traversal, the relations labelling the child nodes of a node  $u$  of  $T$  are semi-joined into the relation labelling  $u$ ; (2) in a top-down traversal, the relation at a node  $u$  is semi-joined into the relation at each child node; (3) all relations are finally joined in yet another bottom-up traversal. The projection  $\pi_U$  is easily integrated into this third traversal. The final result of the query is the resulting relation associated with the root node  $r$  of  $T$ . If this relation is empty after the first bottom-up traversal, then so is the final result and the traversals (2) and (3) can be omitted.

## 3. Experimental Evaluation

Our goal is to shed light on the benefit of realizing structure-guided query evaluation by common RDBMSs. We thus do not want to restrict ourselves to a single architecture or query planning and execution strategy. We have therefore chosen three DBMSs based on different technologies: PostgreSQL 13.4 [14] as a “classical” row-oriented relational DBMS, the column-based in-memory system DuckDB 0.4 [15], and the distributed data processing system Spark SQL 3.3 [16].

We have implemented a proof-of-concept system called YANRE, that works by rewriting a query into a sequence of SQL statements which express Yannakakis’ algorithm. This makes our approach easily portable and we can execute it on the three chosen DBMSs uniformly. The system proceeds in several steps: we first extract the CQ from the given SQL query and

**Table 1**

DuckDB, PostgreSQL, and Spark SQL with or without YANRE over acyclic queries on the MusicBrainz dataset.

Full Enumeration Queries					
Method	Timeouts	Max (s)	Mean (s)	Med. (s)	Std.Dev. (s)
DuckDB	69	770.55	20.38	0.39	77.20
DuckDB+YANRE	<b>29</b>	801.79	24.24	2.03	93.08
PostgreSQL	96	1107.66	45.18	0.78	157.25
PostgreSQL+YANRE	<b>69</b>	786.31	54.81	8.49	120.89
SparkSQL	98	1164.06	79.75	11.03	198.84
SparkSQL+YANRE	<b>35</b>	876.74	114.45	50.87	156.75
Min-Aggregation Queries					
Method	Timeouts	Max (s)	Mean (s)	Med. (s)	Std.Dev. (s)
DuckDB	58	1169.38	23.49	0.26	107.90
DuckDB+YANRE	<b>0</b>	15.57	2.31	1.44	2.38
PostgreSQL	91	1131.08	42.75	0.77	1153.96
PostgreSQL+YANRE	<b>2</b>	236.75	18.01	5.72	29.59
SparkSQL	91	1082.58	94.37	10.59	226.83
SparkSQL+YANRE	<b>0</b>	156.97	31.73	15.95	34.41

transform it into a hypergraph, from which we compute a join tree. We then generate individual SQL statements that correspond to the semi-joins<sup>1</sup> and joins of an execution of Yannakakis' algorithm over the join tree.

We perform experiments using a recent benchmark by Mancini et al. [2], which consists of 435 challenging synthetic join queries over the MusicBrainz dataset [17]. The queries, of which 351 are acyclic, involve between 2 and 30 tables. Classic benchmark datasets, such as TPC-H or TPC-DS, are less interesting for our purposes since their focus is on comparatively small joins.

In Table 1, we summarise our results obtained with two types of experiments for the acyclic queries in [2]. One set of tests (corresponding to SELECT \* FROM ... queries in SQL) is referred to as *full enumeration* queries, which are essentially the original queries of [2], only introducing some projections to lessen the role of unimportant I/O. In a second set of experiments, we explore the effectiveness of computing min-aggregate queries. For this purpose, we transform each query to compute a "MIN" aggregate for a randomly chosen attribute. For such queries Yannakakis' algorithm only needs the first bottom-up traversal, provided that the relation at the root of the join tree contains this attribute<sup>2</sup>.

In Table 1, the Max, Mean, Med. (Median), and Std. Dev. columns provide statistical information for those queries that terminated within the 20 minutes time limit of the respective case. The number of queries that did not terminate within 20 minutes is stated in the Timeouts column. The number of timeouts is significantly lower with YANRE and the remaining timeouts are primarily due to a prohibitively large number of output tuples. Of course, the discrepancy

<sup>1</sup>We express semi-joins as usual via the SQL EXISTS operator.

<sup>2</sup>This can always be achieved by rerooting the jointree.

is particularly big (see particularly the Max column) for the min-aggregate queries, where Yannakakis-style evaluation completely avoids materialisation of all joins.

## 4. Conclusion and Further Work

We have presented a first empirical study of Yannakakis' algorithm in common RDBMSs. Full details are available in [18]. The results, though preliminary, clearly indicate that such a structure-guided approach may indeed allow RDBMSs to handle large join queries that were out of reach before. Our experiments split query evaluation into multiple SQL statements over various temporary tables and naturally leads to some overhead on easy queries, as can be seen in the higher median execution time of YANRE in Table 1. In contrast, queries that are too hard for mainstream RDBMS to solve benefit immensely from evaluation via YANRE. The natural next step is a full-fledged integration of Yannakakis' algorithm into the RDBMSs to get the best of both worlds.

In [18], we have also looked at “almost acyclic” CQs (i.e., CQs of low hypertree-width). Here, many further challenging research questions wait to be solved: above all, database statistics will have to be considered when computing a “good” decomposition – minimising the width alone is no longer enough.

## Acknowledgements

Georg Gottlob is a Royal Society Research Professor and acknowledges support by the Royal Society in this role through the “RAISON DATA” project (Reference No. RP\R1\201074). Matthias Lanzinger acknowledges support by the Royal Society “RAISON DATA” project (Reference No. RP\R1\201074). The work of Cem Okulmus is supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The work of Reinhard Pichler and Alexander Selzer has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013, 10.47379/NXT22018]; and the Christian Doppler Research Association (CDG) JRC LIVE.

## References

- [1] T. Neumann, B. Radke, Adaptive optimization of very large join queries, in: SIGMOD Conference 2018, ACM, 2018, pp. 677–692. URL: <https://doi.org/10.1145/3183713.3183733>. doi:10.1145/3183713.3183733.
- [2] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, A. Ailamaki, Efficient massively parallel join optimization for large queries, in: SIGMOD Conference 2022, ACM, 2022, pp. 122–135. URL: <https://doi.org/10.1145/3514221.3517871>. doi:10.1145/3514221.3517871.
- [3] H. Q. Ngo, E. Porat, C. Ré, A. Rudra, Worst-case optimal join algorithms, J. ACM 65 (2018) 16:1–16:40. URL: <https://doi.org/10.1145/3180143>. doi:10.1145/3180143.
- [4] A. Bonifati, W. Martens, T. Timm, An analytical study of large SPARQL query logs, VLDB J. 29 (2020) 655–679. URL: <https://doi.org/10.1007/s00778-019-00558-9>. doi:10.1007/s00778-019-00558-9.

- [5] W. Fischl, G. Gottlob, D. M. Longo, R. Pichler, Hyperbench: A benchmark and tool for hypergraphs and empirical findings, *ACM J. Exp. Algorithmics* 26 (2021) 1.6:1–1.6:40. URL: <https://doi.org/10.1145/3440015>. doi:10.1145/3440015.
- [6] M. Yannakakis, Algorithms for acyclic database schemes, in: *Proc. VLDB*, 1981, pp. 82–94.
- [7] G. Gottlob, N. Leone, F. Scarcello, Hypertree decompositions and tractable queries, *J. Comput. Syst. Sci.* 64 (2002) 579–627. URL: <https://doi.org/10.1006/jcss.2001.1809>. doi:10.1006/jcss.2001.1809.
- [8] M. Grohe, D. Marx, Constraint solving via fractional edge covers, *ACM Trans. Algorithms* 11 (2014) 4:1–4:20.
- [9] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, C. Ré, Emptyheaded: A relational engine for graph processing, *ACM Trans. Database Syst.* 42 (2017) 20:1–20:44. URL: <https://doi.org/10.1145/3129246>. doi:10.1145/3129246.
- [10] M. Idris, M. Ugarte, S. Vansummeren, The dynamic yannakakis algorithm: Compact and efficient query processing under updates, in: *SIGMOD Conference 2017*, ACM, 2017, pp. 1259–1274. URL: <https://doi.org/10.1145/3035918.3064027>. doi:10.1145/3035918.3064027.
- [11] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, W. Lehner, General dynamic yannakakis: conjunctive queries with theta joins under updates, *VLDB J.* 29 (2020) 619–653. URL: <https://doi.org/10.1007/s00778-019-00590-9>. doi:10.1007/s00778-019-00590-9.
- [12] A. Perelman, C. Ré, Duncemap: Compiling worst-case optimal query plans, in: *SIGMOD Conference 2015*, ACM, 2015, pp. 2075–2076. URL: <https://doi.org/10.1145/2723372.2764945>. doi:10.1145/2723372.2764945.
- [13] S. Tu, C. Ré, Duncemap: Query plans using generalized hypertree decompositions, in: *SIGMOD Conference 2015*, ACM, 2015, pp. 2077–2078. URL: <https://doi.org/10.1145/2723372.2764946>. doi:10.1145/2723372.2764946.
- [14] M. Stonebraker, G. Kemnitz, The postgres next generation database management system, *Commun. ACM* 34 (1991) 78–92. URL: <https://doi.org/10.1145/125223.125262>. doi:10.1145/125223.125262.
- [15] M. Raasveldt, H. Mühleisen, DuckDB: an Embeddable Analytical Database, in: *Proc. SIGMOD 2019*, ACM, 2019, pp. 1981–1984.
- [16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: a unified engine for big data processing, *Commun. ACM* 59 (2016) 56–65. URL: <http://doi.acm.org/10.1145/2934664>. doi:10.1145/2934664.
- [17] MusicBrainz - The Open Music Encyclopedia, <https://musicbrainz.org/>, 2022.
- [18] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, R. Pichler, A. Selzer, Structure-guided query evaluation: Towards bridging the gap from theory to practice, *CoRR abs/2303.02723* (2023). URL: <https://arxiv.org/abs/2303.02723>.