Selective Use of Yannakakis' Algorithm for Consistent Performance Gains

Daniela Böhm[1], Georg Gottlob[2], Matthias Lanzinger[1], Davide Mario Longo[2], Cem Okulmus[3], Reinhard Pichler[1] and Alexander Selzer[1]

[1]TU Wien, Vienna, Austria
[2]University of Calabria, Rende, Italy
[3]Paderborn University, Paderborn, Germany

Abstract

Large intermediate results can cause join queries to run unexpectedly long. This problem is particularly common for analytical queries, which aggregate data over many tables to produce a comparatively small final output, and queries on graph data, where intermediate results blow up quickly. Recent work inspired by Yannakakis' algorithm approaches this by modifying the query engine to avoid materializing unnecessary tuples. However, this requires significant changes to the core of the system, which is not feasible in many situations such as cloud environments or proprietary systems.

In this work, we propose a flexible approach for optimizing long-running join queries from the outside of the DBMS. Rewriting-based realizations of Yannakakis' algorithm suffer from inherent overhead due to the creation of intermediate tables. Thus, we present an approach for detecting and targeting queries which would benefit from a Yannakakis-style optimization. We introduce a new benchmark combining 5 standard benchmarks and augmenting them with additional instances, which provides a sufficient size and diversity for a machine learning based solution. On PostgreSQL, DuckDB and SparkSQL, slowdowns on queries where the rewriting is counterproductive are mostly avoided, as opposed to a naïve application of the rewriting, and we observe significant improvements in end-to-end runtimes over standard query execution and unconditional rewriting.

Keywords

Join Queries, Acyclic Queries, Query Processing

1. Introduction

Join queries can exhibit unexpectedly long runtimes when intermediate results grow significantly larger than the final result. This problem is common in analytical queries, which aggregate data over many tables but intentionally produce compact outputs, and queries over graph data, where the intermediate results explode quickly due to the connectedness of the data.

Yannakakis' algorithm [1] provides a solution to this problem by eliminating all dangling tuples – tuples that will not contribute to the final result of the query – via semi-joins and then computing the join efficiently. In theory, such a strategy of *completely avoiding* the generation of unnecessary intermediate results should always outperform conventional techniques that optimize the join order and thus merely *minimize* intermediate results.

Recent work has brought renewed attention to integrating Yannakakis-style query evaluation into database systems [2, 3, 4, 5, 6]. These approaches modify the query engine to reduce materialization during query execution. However, they require significant changes to the core of the DBMS, which is an option not available in many practical settings, including cloud-hosted databases and proprietary database systems.

DOLAP 2026: 28th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2026, March 24, 2026, Tampere, Finland

daniela.boehm@ymail.com (D. Böhm); georg.gottlob@unical.it (G. Gottlob); matthias.lanzinger@tuwien.ac.at (M. Lanzinger); davidemario.longo@unical.it (D. M. Longo); cem.okulmus@uni-paderborn.de (C. Okulmus); pichler@dbai.tuwien.ac.at (R. Pichler); alexander.selzer@tuwien.ac.at (A. Selzer)
0009-0009-8707-2958 (D. Böhm); 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)

```
SELECT  MIN(c.Id)
FROM    comments AS c, posts AS p, votes AS v, users AS u
WHERE   u.Id = p.OwnerUserId AND u.Id = c.UserId AND
   u.Id = v.UserId AND u.Views>=0 AND p.Score>=0 AND
   p.Score<=28 AND p.ViewCount>=0 AND p.ViewCount<=6517
   AND p.AnswerCount>=0 AND p.AnswerCount<=5 AND
   p.FavoriteCount>=0  AND p.FavoriteCount<=8 AND
   c.CreationDate>='2010-07-27 12:03:40' AND
   p.CreationDate>='2010-07-27 11:29:20' AND
   p.CreationDate<='2014-09-13 02:50:15' AND
   u.CreationDate>='2010-07-27 09:38:05'
```

Figure 1: Slightly modified query 121-097 from the STATS benchmark: original runtime on PostgreSQL (3.38s) vs. Yannakakis-style evaluation (0.11s). If we change the filter condition to `p.FavoriteCount>=8`: original runtime on PostgreSQL (0.05s) vs. Yannakakis-style evaluation (0.09s).

An alternative that avoids these limitations is to realize Yannakakis-style query evaluation via SQL query rewriting [7, 8]. By rewriting a SQL query as a sequence of SQL statements that perform semi-join reductions followed by the final join, any standard DBMS (supporting temporary tables) can be guided to perform Yannakakis-style execution without modifying the core of the system itself.

Rewriting-based approaches, however, introduce overhead due to the execution of multiple SQL statements from outside of the system, and the creation of temporary tables for storing intermediate results. This has the effect that Yannakakis-style rewriting does not always improve the performance, even for queries where the algorithm should in theory be optimal.

The query in Figure 1, a slightly modified version of a query from the STATS benchmark [9], falls into the class of *0MA (zero materialization answerable)* queries [7], which can be evaluated using only semi-joins – completely avoiding the need to compute any joins. Its final result can be retrieved after a single traversal of semi-joins over the join tree. If we consider a join tree of this query with the `comments` relation at the root node (see Figure 2), then we can evaluate this `MIN`-expression after the bottom-up traversal with semi-joins by only considering the resulting relation at the root node.

In theory, the advantage of such a join-less evaluation is clear over conventional query evaluation techniques that first fully evaluate the underlying join query and only then apply the aggregate as a kind of post-processing. For the query in Figure 1, Yannakakis-style evaluation was faster by a factor of roughly 30 times compared to the standard execution of PostgreSQL. However, when changing one of the filter conditions in the WHERE clause from ≥ 0 to ≥ 8 (which, together with the ≤ 8 condition, yields $= 8$), this reverses and the standard execution of PostgreSQL is faster, taking 0.05s compared to the rewriting with 0.09s. This example illustrates a fundamental challenge with Yannakakis-style rewriting in practice: we require a method to decide *when* to apply it. It also shows that making such a decision is non-trivial, since small changes to the query can have significant effects in the effectiveness of Yannakakis-style execution.

In this work, we address this challenge by developing a decision procedure that predicts whether Yannakakis-style rewriting will improve the performance for a given query. We frame this as an algorithm selection problem and solve it using lightweight machine learning models.

To cover various database technologies, we study three diverse DBMSs: PostgreSQL [10] (a row-oriented relational DBMS), DuckDB [11] (a column-oriented in-memory database), and SparkSQL [12] (designed for distributed in-memory processing). We apply a generic SQL-level rewriting approach requiring only minimal variations for the distinct systems (see, e.g., [7, 13, 8, 14, 15]). Taking a SQL-query as input, it is rewritten to an equivalent *sequence of* SQL-statements that guides the DBMS towards Yannakakis-style query evaluation.

Contributions.

- We develop a flexible and robust query rewriting tool based on Apache Calcite. Compared to earlier implementations, it allows for greater applicability to complex queries and straightforward adaptation to new systems and SQL dialects.

- We present a decision procedure for selectively applying the Yannakakis-style query rewriting based on features derived from the query structure as well as DBMS cost estimates. We show that simple models (decision/regression trees) outperform more complex approaches, achieving high accuracy while allowing fine-grained control over the precision-recall tradeoff.
- We introduce a new dataset and benchmark – focusing on hard join queries – which is extensive and diverse enough for effective model training and testing. This new benchmark, which we will refer to as **MEAMBench** (***M**aterialization **E**xplosion **A**ugmented **M**eta **Bench**mark*) is constructed by combining and augmenting queries from several common benchmarks. Beside the algorithm selection problem we are focusing on here, we expect this dataset to be valuable also for other problems, such as query performance prediction.
- To prove the practical potential of the introduced approach, we implement the system **SMASH**, short for ***S**upervised **M**achine-learning for **A**lgorithm **S**election **H**euristics.* SMASH supports multiple DBMSs, including PostgreSQL, DuckDB and SparkSQL, and comprises all steps from feature extraction and algorithm selection up to rewriting.
- Empirical evaluation on MEAMBench clearly confirms significant improvements in end-to-end (e2e) runtimes and decreases in false-positive applications of the rewriting through SMASH compared with the original query evaluation method of the DBMS or unconditionally rewriting.

2. Preliminaries

The queries studied here are either Conjunctive Queries (CQs, i.e., select-project-join queries in Relational Algebra) and, more generally, queries with grouping and aggregation on top:

$$Q = \gamma_{g_1,\dots,g_\ell,\, A_1(a_1),\dots,A_m(a_m)}\big(R_1 \bowtie \cdots \bowtie R_n\big) \tag{1}$$

where $\gamma_{g_1,\dots,g_\ell,\, A_1(a_1),\dots,A_m(a_m)}$ denotes the grouping operation for attributes $g_1, \dots, g_\ell$ and aggregate expressions $A_1(a_1), \dots, A_m(a_m)$. The grouping attributes $g_1, \dots, g_\ell$ are attributes occurring in the relations $R_1, \dots, R_n$, the functions $A_1, \dots, A_m$ are (standard SQL) aggregate functions such as MIN, MAX, COUNT, SUM, AVG, etc., and $a_1, \dots, a_m$ are expressions formed over the attributes from $R_1, \dots, R_n$. We have omitted the projection π_U in Equation (1), since it can be taken care of by the grouping. Moreover, we assume w.l.o.g. that equi-joins have been replaced by natural joins via appropriate renaming of attributes and selections applying to a single relation have been pushed immediately in front of this relation and the R_i's are the result of these selections. A simple query of the form of Equation (1) is given in SQL-syntax in Figure 1.

An *acyclic conjunctive query* (an ACQ, for short) is a CQ $Q = \pi_U(R_1 \bowtie \dots \bowtie R_n)$ that has a *join tree*, i.e., a rooted, labeled tree $\langle T, r, \lambda\rangle$ with root r, such that (1) λ is a bijection that assigns to each node of T one of the relations in $\{R_1, \dots, R_n\}$ and (2) λ satisfies the so-called *connectedness condition*, i.e., if some attribute A occurs in both relations $\lambda(u_i)$ and $\lambda(u_j)$ for two nodes u_i and u_j, then A occurs in the relation $\lambda(u)$ for every node u along the path between u_i and u_j. Actually, the join query underlying the SQL query in Figure 1 is acyclic. A possible join tree is shown in Figure 2. In its original form, *Yannakakis' algorithm* [1] consists of (1) a bottom-up traversal of semi-joins (from child nodes into their parent), (2) a top-down traversal of semi-joins (from a node to its child nodes), and (3) a bottom-up traversal of joins. The result of the query is the final relation associated with the root node r of T. Grouping and the evaluation of aggregates can be carried out as post-processing *after* the evaluation of the join query.

A restricted type of queries of the form given in Equation (1) is the class of *0MA queries* (short for *"zero-materialization answerable"*) *queries* (see [7]). They have to satisfy the following conditions:

- *Guardedness*, meaning that there exists a relation R_i that contains all grouping attributes $g_1, \dots, g_\ell$ and all attributes occurring in the aggregate expressions $A_1(a_1), \dots, A_m(a_m)$. Then R_i is called the *guard* of the query. If several relations satisfy this property, we arbitrarily choose one guard.

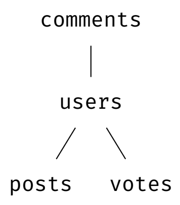

Figure 2: Join tree for the query in Fig. 1

- *Set-safety*: we call an aggregate function *set-safe*, if it is invariant under duplicate elimination, i.e., its value over any set S of tuples remains unchanged if duplicates are eliminated from S. A query satisfies the set-safety condition, if all its aggregate functions $A_1 \dots, A_m$ are set-safe.

The root of the join tree can be arbitrarily chosen. For a 0MA query, we choose the node labeled by the guard as the root node. Hence, if all aggregate functions are set-safe (i.e., multiplicities do not matter), then we can apply the grouping and aggregation $\gamma_{g_1,\dots,g_\ell,\, A_1(a_1),\dots,A_m(a_m)}$ to the relation at the root node right after the first bottom-up traversal. In SQL, in particular, the `MIN` and `MAX` aggregates are inherently set-safe. Moreover, an aggregate becomes set-safe when combined with the `DISTINCT` keyword. For instance, `COUNT DISTINCT` is a set-safe aggregate function.

An example of a 0MA query is given in Figure 1: it is trivially guarded (i.e., there is no grouping and the only aggregate expression is over a single attribute) and set-safe (since the only aggregate function in this query is `MIN`).

3. Related Work

Acyclic queries and Yannakakis-style query evaluation. Yannakakis' algorithm [1] has recently received renewed attention for the optimization of hard join queries. Several works have aimed at bringing its advantages into DBMSs from the outside via SQL query rewriting [7, 8, 14, 13], and similar methods such as generating Scala code expressing Yannakakis' algorithm as Spark RDD-operations [16]. An important line of research in this area has been concerned with the integration of ideas of Yannakakis-style query evaluation into DBMSs while avoiding the overhead of *several* traversals of the join tree via semi-joins and joins [17, 18, 2, 13]. It should however be noted that, despite all the progress made in integrating and fine-tuning Yannakakis' algorithm, we are still left with the fact that this optimization leads to a performance improvement in *some* but *not all* cases. Indeed, not even for 0MA queries (as one of the simplest forms of acyclic queries), an improvement in *all* cases is guaranteed, as will be confirmed by our empirical study presented in Section 5.

Decompositions. In order to go beyond acyclic queries, a major area of research seeks to extend Yannakakis-style query answering to "almost-acyclic" queries via various notions of decompositions and their associated width measures, such as hypertree-width, soft hypertree-width, generalized hypertree-width, and fractional hypertree-width [19, 14, 20, 21]. Several implementations [22, 16, 23, 24] combine Yannakakis-style query execution with worst-case optimal joins [25]. To address the problem of minimal-width decompositions not necessarily being cost-optimal, approaches of integrating statistics about the data into the search for the best decomposition have been proposed and implemented [14, 26].

Query rewriting. Optimizing queries before they enter the DBMS is a different strategy towards query optimization that has been successfully applied in standard DBMSs [27, 15]. Although DBMSs already perform optimizations on the execution of the query, it has been shown that rewriting the query itself can still be highly effective. The WeTune [28] system goes even further, and can be used to automatically discover rewrite rules but comes with the disadvantage of extremely long runtimes.

Machine learning for databases. There has been growing interest in the application of machine learning techniques to improve the performance of database systems [29]. Many systems have been presented, of which we list some of the most relevant to this work. Bao [30, 31] learns when to apply query hints. Neo [32] uses machine learning for end-to-end query optimization. SkinnerDB [33] applies reinforcement learning to adaptive query optimization.

Query performance prediction. Predicting the performance of a query – usually the runtime, or sometimes the resource requirements – is related to the problem of deciding whether to rewrite a query. Runtime prediction has been performed by constructing cost models based on statistical information of the data [34], on SQL queries [35], and XML queries [36]. Further approaches use machine learning and deep learning to predict the runtimes of single queries [37, 38, 39] or concurrent queries (workload performance prediction) [40, 41].

4. Selectively Applying Yannakakis-style Query Evaluation

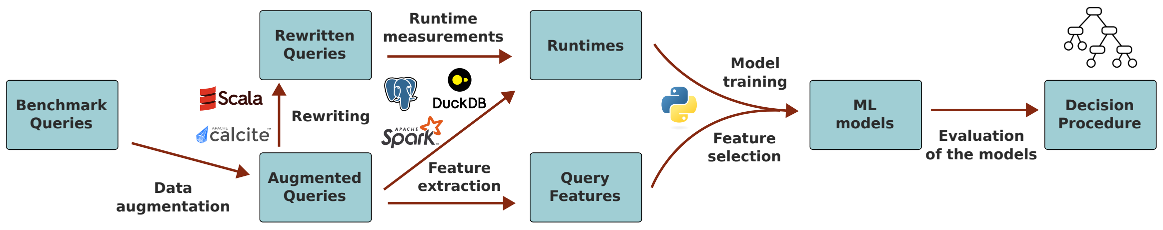

Figure 3: Methodology workflow.

As illustrated in Section 1, a Yannakakis-style rewriting does not always improve the performance, even for queries where it would be expected based on the reduction in intermediate results. To benefit from the rewriting in practice, it is clear that we require a decision procedure for the selective application of the rewriting. We are thus faced with an algorithm selection problem, where we have to decide, for every database instance and query, which query evaluation method should be applied. In this section, we describe our approach towards solving this algorithm selection problem. We develop a pipeline, starting from the databases and queries, and resulting in a decision procedure. In the process of this, we create a new implementation for query rewriting using Apache Calcite. An overview of the workflow is given in Figure 3.

Robust and Flexible Yannakakis-style Query Rewriting We improve upon the approach of [7]. This approach involves rewriting each query into an (output-)equivalent sequence of SQL queries which "guides" the DBMSs to apply a Yannakakis-style evaluation strategy.

We take a SQL query as input and first transform it into a hypergraph. The approach of [7] begins by applying the GYO-reduction [42]. It is thus verified that the CQ is acyclic and, if so, a join tree is constructed. In the case of 0MA queries, containing an aggregate expression with a single attribute, we choose this relation as the root of the join tree, allowing us to simplify Yannakakis' algorithm by skipping the final two traversals. The rewriting proceeds by creating a sequence of SQL queries. The tree is traversed in a bottom-up fashion, starting from the leaves, producing auxiliary tables via `CREATE VIEW` statements, for the original relations, and `CREATE UNLOGGED TABLE`, for the intermediate results. For 0MA queries the aggregation is performed last, after the creation of auxiliary tables. The use of `UNLOGGED TABLE` is an optimization for PostgreSQL to reduce overhead of the write-ahead log.

We illustrate the query rewrite process for the 0MA query q, which is shown, with its join tree, in Figure 4. Note that, in this example, for each relation in the query, as well as its associated filters (involving only attributes in the relation), a view is created. Overhead due to this can be slightly reduced in practice by integrating the selections and filters of the `CREATE VIEW` statements into the `CREATE UNLOGGED TABLE` statements.

The implementation makes use of Apache Calcite [43], a modular framework for query optimization. Schema information, which is required to transform the query into a logical plan reliably (for example,

```
q: SELECT MIN(u.Id) FROM votes as v, badges as b, users as u
   WHERE u.Id=v.UserId AND v.UserId=b.UserId
      AND v.BountyAmount>=0 AND v.BountyAmount<=50 AND u.DownVotes=0
```

```
qr1: CREATE VIEW E3 AS
  SELECT * FROM users
  WHERE DownVotes = 0

qr2: CREATE VIEW E2 AS
  SELECT * FROM badges

qr3: CREATE UNLOGGED TABLE
  E3E2 AS SELECT * FROM E3
  WHERE EXISTS (SELECT 1
    FROM E2
    WHERE E3.Id=E2.UserId)
```

```
qr4: CREATE VIEW E1 AS
  SELECT * FROM votes
  WHERE BountyAmount >= 0
    AND BountyAmount <= 50

qr5: CREATE UNLOGGED TABLE
  E3E2E1 AS
  SELECT MIN(Id) AS EXPR$0
  FROM E3E2 WHERE EXISTS
    (SELECT 1 FROM E1
     WHERE E3E2.Id=E1.UserId)

qr6: SELECT * FROM E3E2E1
```

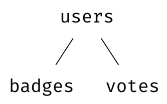

Figure 4: Example rewriting of a 0MA query q (top) and its join tree (bottom right). The rewritten sequence q_{r1}–q_{r6} (bottom left) performs semi-join reductions following the join tree structure before computing the aggregate.

if attributes are referred to implicitly, the schema is required to resolve the reference), is extracted directly from the DBMS over the JDBC interface. The logical plan resulting from the SQL string is next transformed into a convenient intermediate representation via query optimizer rules, where all filters are pushed to the bottom of the plan, which allows us to retrieve a set of subtrees. These subtrees of the plan, corresponding to the hyperedges, are followed by equi-join operator nodes, from which the hypergraph can be constructed. The approach of using the parser and optimizer of Apache Calcite, as opposed to earlier works [7], which go directly from parsing a query to a rewriting, provides multiple advantages. Beside being significantly more robust and supporting complex queries and SQL features, Apache Calcite supports a large amount of SQL dialects, making it straightforward to extend the rewriting to various systems. The implementation of the rewriting, together with information on feature extraction and model training, can be found here: https://github.com/dbai-tuw/yannakakis-rewriting.

Benchmark Data and Data Augmentation When constructing our new dataset MEAMBench, we pursue two major goals to make the dataset suitable for model training and testing: it should be sufficiently big and sufficiently diverse. We thus *select and adapt common benchmarks* focusing on join queries and choose queries to which the optimization technique is applicable (acyclic equi-join queries). To address the diversity aspect, we collect diverse sets of data and queries from different domains and designed for different purposes. Thus, as a basis, we have chosen several widely used benchmarks, namely *JOB (Join Order Benchmark)* [44], *STATS/STATS-CEB* [9], four different datasets (namely cit-Patents, wiki-topcats, web-Google and com-DBLP) from *SNAP (Stanford Network Analysis Project)* [45] together with synthetic queries introduced in [46], *LSQB (Large-Scale Subgraph Query Benchmark)* [47], and *HETIONET* [48].

We focus on ACQs, which make up the vast majority of the queries in the base benchmarks. The number of (acyclic) CQs of each dataset is given in Table 1. Some of the queries in the benchmarks are enumeration queries and some already contain some aggregate (in particular, `MIN`) and satisfy the 0MA conditions. Clearly, also from the enumeration queries, we can derive 0MA queries by an appropriate aggregate expression (again, in particular, with the `MIN` aggregate function) in the SELECT clause of the query. For this, we randomly choose a table occurring in the query and one column of this table. Our collection of data and queries from different benchmarks results in 219 acyclic queries. This is still not sufficient for training and testing ML models. We, therefore, perform the following steps for data augmentation: “filter augmentation” (for all queries) followed by “aggregate-attribute augmentation” (for 0MA-queries) and “enumeration augmentation” (for enumeration queries), respectively.

Through the “filter augmentation” we produce duplicates of all queries having filters (i.e., selection

Table 1
Overview of the 0MA and enumeration queries after augmentation. In total, we get 4677 queries, consisting of 2936 0MA queries and 1741 enum queries.

Dataset	# ACQs	+ Filter	+ Filter & Agg	+ Filter & Enum
STATS	146	432	1876	1264
SNAP	40	40	244	120
JOB	15	45	264	135
LSQB	2	2	14	6
HETIONET	26	72	538	216
Total	219	591	2936	1741

conditions on a single table) by changing some filters in a way that the sizes of the resulting relations vary between these queries. In fact, the SNAP and LSQB queries do not have filter conditions, which means that there is no filter augmentation for them. For 0MA queries, we next apply the "aggregate-attribute augmentation" to vary the table from which we take the MIN-attribute. This is done in a way that every table occurring in the query appears once in the MIN-expression. For enumeration queries, we randomly choose two of the attributes used in join conditions and write them into the SELECT clause of the query. In summary, after applying the data augmentation step, we have obtained 4677 queries in total (see Table 1). The MEAMBench benchmark can be found here: https://github.com/dbai-tuw/MEAMBench.

Feature Extraction and Selection We choose a variety of features that we derive from the structure of the query itself, from the join tree constructed in the process of rewriting the query, and from statistics determined by PostgreSQL or DuckDB over the database. The latter kind of features is extracted from the query optimizer's estimates, and obtained via the EXPLAIN command. Note that Spark SQL does not provide an EXPLAIN command. However, we will explain below how to circumvent this shortcoming. Another challenge are features that are based on a set, of variable length, containing numeric values. In order to reduce such a set of values into a fixed-length list of values, we calculate, for each set, several statistics: min, the 0.25-quantile (referred to as q25), median, the 0.75-quantile (referred to as q75), max, and mean. In the list of features below, we use "*" (e.g. **B7***) to mark which features consist of variable-length sets, and hence will get reduced to the mentioned collection of 6 values.

Features derived from the query. The following features are easily obtained by inspecting the query itself:
Feature B1: *is 0MA?* indicates (1 or 0) if the query is 0MA,
Feature B2: *number of relations*,
Feature B3: *number of conditions*, the number of (in)equality conditions in the WHERE clause,
Feature B4: *number of filters*, the number of (in)equality conditions occurring in the query, and
Feature B5: *number of joins.*

Features based on the join tree. These features are inspired by the work in [49] on tree decompositions:
Feature B6: *depth*, which is the maximal distance between the root of the used join tree and a leaf node,
Feature B7*: *container counts*, a set of numbers, for each variable, the number of join tree nodes containing it. This measure indicates how many relations are joined on the same variable, and
Feature B8*: *branching degrees*, a set indicating for each node the number of children it has.

These eight features (B1)-(B8*) are the shared *"basic features"*. In addition, we can use statistical information from the database and the estimates for the query evaluation, though the exact features that are exposed differs between DBMSs. In case of PostgreSQL and DuckDB, we have the EXPLAIN command at our disposal to obtain relevant further information. For PostgreSQL, we thus select the following additional features, which we refer to as *"PSQL features"*:
Feature P1: *estimated total cost* (of the query),
Feature P2*: *estimated single table rows*, which stands for the estimated number of rows for each table involved in the query *after* the application of the filter conditions, and

Feature P3*: *estimated join rows*, the estimated number of rows of each join *before* the application of the filter conditions.

The EXPLAIN command for DuckDB behaves differently. It allows us to derive a single *"DDB feature"*:
Feature D1*: *estimated cardinalities*, the estimated number of rows after each node in the logical plan, such as filters and joins.

As Spark SQL (the open-source version) does not perform cost-based optimization, it also does not keep track of statistical information about the data, and cannot estimate cardinalities or costs of a plan. However, since SparkSQL also does not provide a persistent storage layer, tables are commonly imported from another database via JDBC. This implies that, in practice, the statistical features can easily be extracted from this database and used for the decision whether to rewrite the query in SparkSQL. For the experiments presented in Section 5, we extracted these features from PostgreSQL, hence SparkSQL has the same feature set as PostgreSQL.

Model Training and Evaluation Our ultimate goal is the development of a decision procedure, for making a binary decision between two evaluation methods for acyclic queries. Hence, we are dealing with a *classification problem* with *2 possible outcomes*. We will refer to these two possible outcomes as 0 vs. 1 to denote the original evaluation method of the DBMS vs. a Yannakakis-style evaluation (enforced by our query rewriting). Alternatively, we can first address a *regression problem* that predicts the runtime difference between the two query evaluation techniques and then classify a query as 0 (if the predicted value suggests faster evaluation by the original method of the DBMS within a certain threshold) or 1 (otherwise). We will come back to the selection of an appropriate threshold in Section 5.

We have considered 7 Machine Learning model types for our algorithm selection problem, namely k-nearest neighbors (k-NN), decision tree, random forest, support vector machine (SVM), and 3 forms of neural networks (NNs): multi-layer perceptron (MLP), hypergraph neural network (HGNN) and a combination of the two. MLP is the “classical” deep neural network type. Hypergraph neural networks [50], are less known. With their idea of representing the hypergraph structure in a vector space, HGNNs seem well suited to capture structural aspects of CQs.

After running the 4677 queries on the 3 selected DBMSs, we have to prepare the input data for training the ML models of the 7 types mentioned above. For our supervised learning tasks (classification and regression), we have to label each feature vector for each of the 3 DBMSs. The labeled data can then be split into training data, validation data, and test data – choosing a common ratio of 80:10:10. To get more accurate results, we do 10-fold cross validation. That is, we split the 90% of the data that were chosen for training and validation in 10 different ways in a ratio 80:10 into training:validation data and, thus, repeat the training-validation step 10 times.

In order to ultimately choose the “best” model for our decision procedure, we evaluate, for every feature vector, the predicted classification compared to the optimal decision. This leads to 4 possible outcomes comparing between predicted and actual value, namely TP (true positive) and TN (true negative) for correct classification and FP (false positive) and FN (false negative) for misclassification. We consider 3 common metrics: accuracy, precision, and recall. The natural goal when selecting a particular model is to maximize the accuracy. However, in our context, we consider the precision equally important. That is, we find it particularly important to minimize false positives, i.e., in case of doubt, it is better to stick to the original evaluation method of the DBMS rather than wrongly choosing an alternative method. Apart from the purely *quantitative* assessment of a model in terms of accuracy, precision, and recall, we also carry out a *qualitative* analysis. That is, for each of the misclassified cases, we investigate by how much the chosen evaluation method is slower than the optimal method – again with a focus on false positive classifications.

5. Experimental Results

We now present the experimental results obtained by putting the algorithm selection as described in Section 4 to work. With these experiments, we are aiming to answer the following key questions:

Q1 How well can machine learning methods predict whether Yannakakis-style evaluation is preferable over standard execution?

Q2 Can we use these machine learning models to gain insights about the circumstances in which Yannakakis-style query evaluation is preferable over standard query execution?

Q3 How well does good algorithm selection performance translate to query evaluation times on different DBMSs?

Q4 To what extent can we optimize for precision while maintaining e2e runtime and accuracy?

Table 2
Performance of Machine Learning Classifiers on the PostgreSQL runtimes. We show accuracy, precision and recall for binary classifiers that predict whether rewriting to Yannakakis style evaluation leads to performance gain.

Algorithm	**0MA Queries**			**Acyc. Queries**		
	Acc. (%)↑	Prec. ↑	Rec. ↑	Acc. (%) ↑	Prec. ↑	Rec. ↑
Decision Tree	**0.94**	**0.92**	**0.97**	**0.95**	**0.95**	0.92
Random forest	**0.94**	**0.92**	**0.97**	**0.95**	0.94	**0.93**
k-NN	0.91	0.91	0.90	0.91	0.88	0.91
SVM	0.85	0.85	0.84	0.84	0.82	0.77
MLP	0.87	0.89	0.86	0.85	0.84	0.77
HGNN	0.83	0.84	0.85	0.79	0.70	0.75
HGNN+MLP	0.82	0.78	0.93	0.81	0.77	0.72

Table 3
Mean average error (MAE) and algorithm selection accuracy for regression models predicting the difference between the runtime of the original and the rewritten query.

Algorithm	**Acyc. Queries**			
	MAE	Acc.	Prec.	Rec.
Decision Tree	**0.06**	**0.96**	**0.96**	**0.94**
Random forest	0.08	0.95	0.94	0.93
k-NN	0.18	0.91	0.88	0.91
SVM	0.61	0.79	0.76	0.72
MLP	0.32	0.81	0.72	0.77
HGNN	0.48	0.71	0.57	0.85

Model Performance We compare the performance of various learned models in terms of accuracy, precision, and recall (see Tables 2 and 3). We do this for all queries (i.e., 0MA and enumeration) based on the feature vectors and the runtime data obtained with the 3 chosen DBMSs. It turns out that the simplest models (decision tree and random forest) perform best, with accuracy, precision and recall values well above 0.90.

Fine-tuning Precision and Recall So far we have only considered choosing a *threshold* of the predicted runtime difference of the regression model at 0 (i.e., if the predicted runtime difference is below this threshold, we choose Yannakakis-style query evaluation, and, otherwise, we opt for the original query execution). However, we can make use of the regression model as a useful tool to configure the trade-off between precision and recall, depending on the requirements of the application. To simplify the presentation, we focus on one DBMS and on one model, namely PostgreSQL and decision trees. The extension to other models and DBMSs is straightforward and yields similar results. In Figure

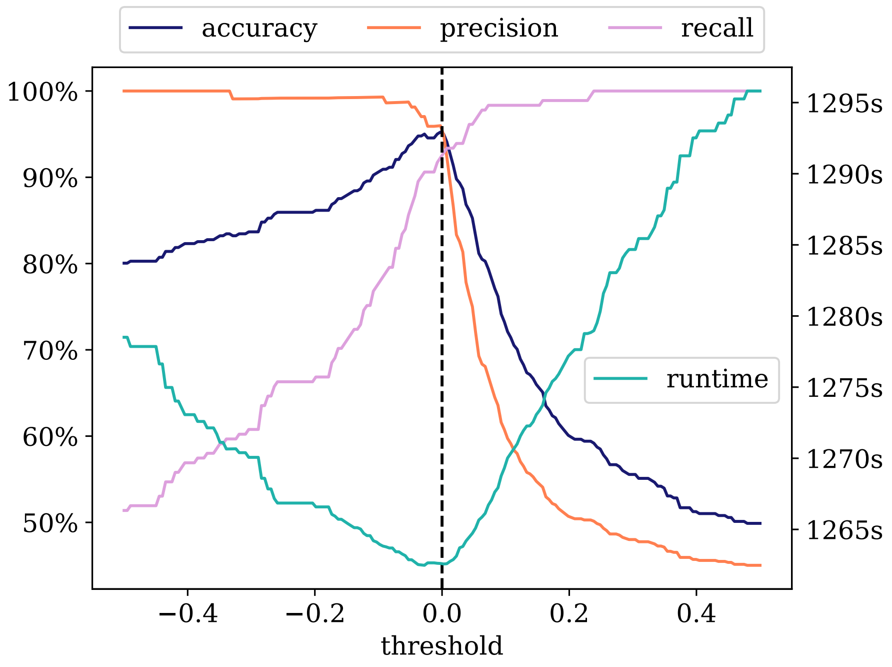

Figure 5: Accuracy, precision, recall, as well as the e2e runtime over the test set, of the regression model converted to a classifier, depending on the threshold set (decision tree regression, PostgreSQL, all queries).

5, we show how changing the threshold affects the accuracy, precision and recall of the resulting classification model. Clearly, the maximum accuracy is at the 0-threshold. The accuracy, however, continues to be high in the direction of negative thresholds, while falling off quickly in the direction of a positive threshold. In particular, this shows that an optimization of precision can be performed without sacrificing much recall.

To summarize: Table 2 and Table 3 show very high accuracy, precision and recall for decision trees, which allows us to positively answer the key question **Q1**. In Figure 5 we also get an answer for our key question **Q4**: by choosing the right threshold one can achieve almost perfect precision, with only modest reductions in accuracy and e2e runtime.

Insights from Decision Trees It is particularly interesting that decision trees are the top-performing models as they are highly interpretable and can provide us with insights into the features that strongly affect the prediction. To get a deeper understanding of the impact that the various features have on the outcome of the decision tree model, we have analyzed their Gini coefficients [51], which, intuitively, measure how much each feature affects the prediction. Interestingly, there were quite significant differences between the tested DBMSs. On PostgreSQL, the estimated join size is far more significant than the questions of whether we are dealing with a 0MA query or an enumeration query. In contrast, on DuckDB, the latter distinction has the biggest impact. Full details of the Gini coefficients of the most important features on each DBMS are given in the extended paper [52]. This ability of decision trees to highlight which specific features affect the prediction the most, helps us answer the key question **Q2** positively.

Effects on Database Performance Based on the training and evaluation of various ML models, we choose the decision tree regression model followed by classification with threshold value 0 for our runtime measurements. As mentioned in the introduction, we refer to our method, that decides whether to rewrite to Yannakakis-style evaluation based on the prediction of the decision tree model as SMASH, short for ***S****upervised* ***M****achine-learning for* ***A****lgorithm* ***S****election* ***H****euristics*. We have measured these runtimes only on the test set queries, i.e., *no queries from these experiments were seen by the model*

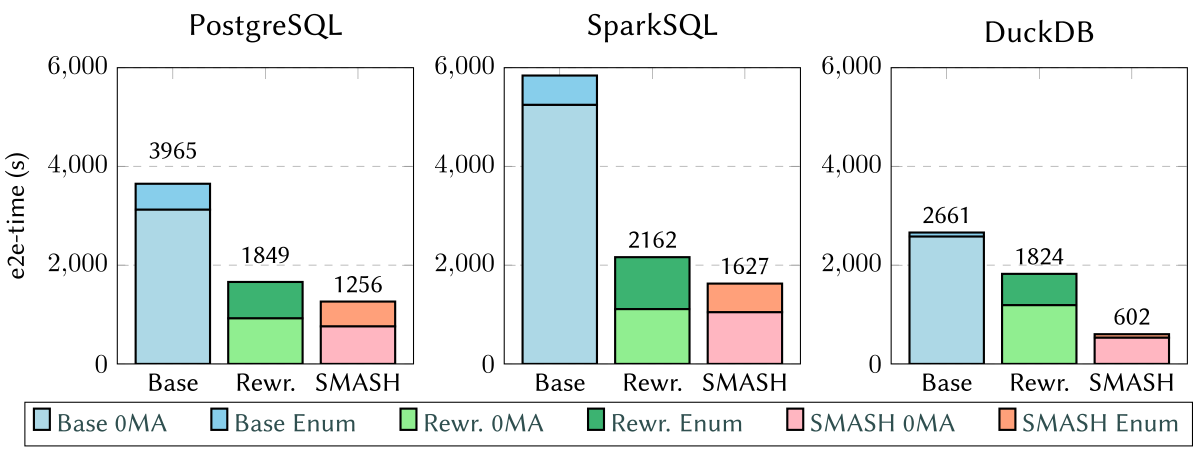

Figure 6: Comparison of e2e performance over the test set queries for three database systems: PostgreSQL, Spark (via SparkSQL), and DuckDB. The full bar indicates the runtime over all test set queries, the lower mark indicates the time for 0MA queries.

at training time. The whole evaluation is performed on a server with 128GB RAM, a 16-core AMD EPYC-Milan CPU, and a 100GB SSD disk, running Ubuntu 22.04.2 LTS. After a warm-up run, the original query, as well as the rewritten version, is evaluated five times, and then we report the mean of those five runs. Fundamentally, we are interested in improving overall query answering performance. We investigate this by analyzing the e2e runtime necessary to answer all queries in the test set, i.e., the time to run all the benchmark queries cumulatively. This includes the time for the algorithm selection included in SMASH, which was around 2 milliseconds in total for all queries together.

We summarize our analysis in Figure 6, where *Base* refers to the baseline of executing the queries directly in the DBMS, *Rewriting* refers to unconditionally applying the rewriting for Yannakakis-style evaluation, and SMASH refers to the use of our algorithm selection model as described above. To study the robustness of our approach we perform these experiments on three different DBMSs: PostgreSQL, Spark (via SparkSQL), and DuckDB. The significant technical differences between the three systems provide us with a way to study the performance of our method independently of specific DBMS technologies. We report timeouts as follows: if only one of the evaluation methods (rewriting or base case) times out, then we report in Figure 6 for such queries as runtime the value of the timeout (= 100s). On the other hand, if both evaluation methods time out, we exclude them from the comparison, since algorithm selection cannot affect anything in such a case. Out of the 441 queries involved in the test set, there were 27 queries that timed out for both evaluation methods on PostgreSQL, 13 queries that timed out for both evaluation methods on SparkSQL, and 30 queries that timed out for both evaluation methods on DuckDB.

Consistently, over all systems, we can observe a large improvement of algorithm selection over the other two alternatives. Furthermore, we see that even when unconditionally rewriting, there are significant improvements over the baseline execution for all three systems tested. However, this improvement comes from speedups specifically on queries that are hard for traditional RDBMS execution. These large improvements offset more common minor slowdowns using the *Rewriting* approach. Using SMASH we are able to achieve **the best of both worlds**: major speedups on hard queries without the minor performance degradations. In summary, we can give a strongly positive answer to key question **Q3**. Our algorithm selection method clearly improves the e2e query evaluation times both significantly and consistently over different DBMSs (each with their own trained model).

The Effect of Data Augmentation. We explore the impact of data augmentation on model performance through a focused ablation study, carefully examining the effectiveness of our augmentation strategies. Specifically, we compare models trained on two distinct training sets: one encompassing the fully augmented dataset derived from the complete MEAMBench, and another restricted solely to *base* queries with all augmented data removed. Table 4 succinctly summarizes this comparison, presenting accuracy, precision, and recall metrics for both the "Base" set comprised of 0MA and enumeration queries without filter augmentation, and the full augmented training set. Our analysis reveals that data augmentation substantially enhances accuracy and recall, with even greater gains in precision.

Table 4
Ablation study comparing the performance of a model on a training set with augmented data to one without. Values are based on the evaluation of the regression model with a 0.5-threshold on the test set.

	Base			Augmented		
DBMS	Acc.	Prec.	Rec.	Acc.	Prec.	Rec.
Postgres	0.88	0.83	0.89	0.95	0.95	0.93
DuckDB	0.86	0.79	0.84	0.91	0.88	0.87
SparkSQL	0.85	0.79	0.81	0.93	0.92	0.88

6. Conclusion

In this work we addressed the problem of slow join queries caused by large intermediate results. Yannakakis-style query evaluation offers a solution by eliminating dangling tuples, but recent approaches require modifications to the DBMS internals, which is an option unavailable in many situations.

We thus considered an alternative based on SQL rewriting, which is applicable to any standard DBMS. However, due to the overhead of the external rewriting, it does not always improve performance, even for queries with unnecessary intermediate results. To unlock the potential of the Yannakakis-style rewriting, we developed SMASH, a decision procedure that predicts when rewriting is beneficial. Our experiments on PostgreSQL, DuckDB, and SparkSQL demonstrate significant improvements in e2e runtimes over both standard execution and unconditional rewriting, while largely avoiding slowdowns on queries where the rewriting is counterproductive.

MEAMBench, our new benchmark combining five standard benchmarks extended via data augmentation, focuses on hard join queries and provides sufficient size and diversity for training learned models.

Although we focused on acyclic and 0MA queries, the methodology we propose is broadly applicable. It opens the door to more intelligent optimization strategies across a wide range of techniques, including decomposition-based methods for cyclic queries.

Acknowledgments

This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013]. Georg Gottlob and Davide Mario Longo acknowledge support from the PNRR FAIR project Future AI Research (PE00000013), Spoke 9 - Green-aware AI, under the NRRP MUR program funded by the NextGenerationEU initiative.

Declaration on Generative AI

During the preparation of this work, the authors used Claude and ChatGPT in order to: Grammar and spelling check, paraphrase and reword. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

[1] M. Yannakakis, Algorithms for acyclic database schemes, in: Proceedings of the 7th International Conference on Very Large Databases, VLDB 1981, Cannes, VLDB, 1981, pp. 82–94.

[2] A. Birler, A. Kemper, T. Neumann, Robust join processing with diamond hardened joins, Proc. VLDB Endow. 17 (2024) 3215–3228. URL: https://www.vldb.org/pvldb/vol17/p3215-birler.pdf.

[3] L. Bekkers, F. Neven, S. Vansummeren, Y. R. Wang, Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores, Proc. VLDB Endow. 18 (2025) 2413–2426. URL: https://www.vldb.org/pvldb/vol18/p2413-vansummeren.pdf. doi:10.14778/3742728.3742737.

[4] M. Lanzinger, R. Pichler, A. Selzer, Avoiding materialisation for guarded aggregate queries, Proc. VLDB Endow. 18 (2025) 1398–1411. URL: https://www.vldb.org/pvldb/vol18/p1398-selzer.pdf.

[5] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, H. Zhang, Debunking the myth of join ordering: Toward robust SQL analytics, Proc. ACM Manag. Data 3 (2025) 146:1–146:28. URL: https://doi.org/10.1145/3725283. doi:10.1145/3725283.

[6] P. Koutris, S. Vansummeren, Q. Wang, Y. R. Wang, X. Yu, Database theory in action: Yannakakis' algorithm, CoRR abs/2601.00098 (2026). URL: https://doi.org/10.48550/arXiv.2601.00098. doi:10.48550/ARXIV.2601.00098. arXiv:2601.00098.

[7] 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://doi.org/10.48550/arXiv.2303.02723. doi:10.48550/arXiv.2303.02723. arXiv:2303.02723.

[8] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, R. Pichler, A. Selzer, Reaching back to move forward: Using old ideas to achieve a new level of query optimization (short paper), in: B. Kimelfeld, M. V. Martinez, R. Angles (Eds.), Proceedings of the 15th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2023), Santiago de Chile, Chile, May 22-26, 2023, volume 3409 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3409/paper6.pdf.

[9] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, B. Cui, Cardinality estimation in DBMS: A comprehensive benchmark evaluation, Proc. VLDB Endow. 15 (2021) 752–765. URL: https://www.vldb.org/pvldb/vol15/p752-zhu.pdf. doi:10.14778/3503585.3503586.

[10] 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.

[11] M. Raasveldt, H. Mühleisen, DuckDB: an Embeddable Analytical Database, in: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, ACM, 2019, pp. 1981–1984.

[12] 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.

[13] Q. Wang, B. Chen, B. Dai, K. Yi, F. Li, L. Lin, Yannakakis+: Practical acyclic query evaluation with theoretical guarantees, CoRR abs/2504.03279 (2025). URL: https://doi.org/10.48550/arXiv.2504.03279. doi:10.48550/ARXIV.2504.03279. arXiv:2504.03279.

[14] M. Lanzinger, C. Okulmus, R. Pichler, A. Selzer, G. Gottlob, Soft and constrained hypertree width, Proc. ACM Manag. Data 3 (2025) 114:1–114:25. URL: https://doi.org/10.1145/3725251. doi:10.1145/3725251.

[15] X. Zhou, G. Li, J. Wu, J. Liu, Z. Sun, X. Zhang, A learned query rewrite system, Proc. VLDB Endow. 16 (2023) 4110–4113. URL: https://www.vldb.org/pvldb/vol16/p4110-li.pdf. doi:10.14778/3611540.3611633.

[16] B. Dai, Q. Wang, K. Yi, Sparksql+: Next-generation query planning over spark, in: S. Das, I. Pandis, K. S. Candan, S. Amer-Yahia (Eds.), Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023, ACM, 2023, pp.

115–118. URL: https://doi.org/10.1145/3555041.3589715. doi:10.1145/3555041.3589715.

[17] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, H. Zhang, Debunking the myth of join ordering: Toward robust SQL analytics, CoRR abs/2502.15181 (2025). URL: https://doi.org/10.48550/arXiv.2502.15181. doi:10.48550/ARXIV.2502.15181. arXiv:2502.15181.

[18] Y. Yang, H. Zhao, X. Yu, P. 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.

[19] 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.

[20] I. Adler, G. Gottlob, M. Grohe, Hypertree width and related hypergraph invariants, Eur. J. Comb. 28 (2007) 2167–2181. URL: https://doi.org/10.1016/j.ejc.2007.04.013. doi:10.1016/j.ejc.2007.04.013.

[21] M. Grohe, D. Marx, Constraint solving via fractional edge covers, ACM Trans. Algorithms 11 (2014) 4:1–4:20.

[22] 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.

[23] A. Perelman, C. Ré, Duncecap: Compiling worst-case optimal query plans, in: T. K. Sellis, S. B. Davidson, Z. G. Ives (Eds.), Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, ACM, 2015, pp. 2075–2076. URL: https://doi.org/10.1145/2723372.2764945. doi:10.1145/2723372.2764945.

[24] S. Tu, C. Ré, Duncecap: Query plans using generalized hypertree decompositions, in: T. K. Sellis, S. B. Davidson, Z. G. Ives (Eds.), Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, ACM, 2015, pp. 2077–2078. URL: https://doi.org/10.1145/2723372.2764946. doi:10.1145/2723372.2764946.

[25] 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.

[26] F. Scarcello, G. Greco, N. Leone, Weighted hypertree decompositions and optimal query plans, in: C. Beeri, A. Deutsch (Eds.), Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France, ACM, 2004, pp. 210–221. URL: https://doi.org/10.1145/1055558.1055587. doi:10.1145/1055558.1055587.

[27] X. Zhou, G. Li, C. Chai, J. Feng, A learned query rewrite system using monte carlo tree search, Proc. VLDB Endow. 15 (2021) 46–58. URL: http://www.vldb.org/pvldb/vol15/p46-li.pdf. doi:10.14778/3485450.3485456.

[28] Z. Wang, Z. Zhou, Y. Yang, H. Ding, G. Hu, D. Ding, C. Tang, H. Chen, J. Li, Wetune: Automatic discovery and verification of query rewrite rules, in: Z. G. Ives, A. Bonifati, A. E. Abbadi (Eds.), SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022, ACM, 2022, pp. 94–107. URL: https://doi.org/10.1145/3514221.3526125. doi:10.1145/3514221.3526125.

[29] X. Zhou, C. Chai, G. Li, J. Sun, Database meets artificial intelligence: A survey, IEEE Trans. Knowl. Data Eng. 34 (2022) 1096–1116. URL: https://doi.org/10.1109/TKDE.2020.2994641. doi:10.1109/TKDE.2020.2994641.

[30] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making learned query optimization practical, in: G. Li, Z. Li, S. Idreos, D. Srivastava (Eds.), SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, ACM, 2021, pp. 1275–1288. URL: https://doi.org/10.1145/3448016.3452838. doi:10.1145/3448016.3452838.

[31] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making learned query optimization practical, SIGMOD Rec. 51 (2022) 6–13. URL: https://doi.org/10.1145/3542700.3542703. doi:10.1145/3542700.3542703.

[32] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A learned query optimizer, Proc. VLDB Endow. 12 (2019) 1705–1718. URL: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf. doi:10.14778/3342263.3342644.

[33] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, A. Rayabhari, Skinnerdb:

Regret-bounded query evaluation via reinforcement learning, ACM Trans. Database Syst. 46 (2021) 9:1–9:45. URL: https://doi.org/10.1145/3464389. doi:10.1145/3464389.

[34] B. He, I. Ounis, Query performance prediction, Inf. Syst. 31 (2006) 585–594. URL: https://doi.org/10.1016/j.is.2005.11.003. doi:10.1016/J.IS.2005.11.003.

[35] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, J. F. Naughton, Predicting query execution time: Are optimizer cost models really unusable?, in: C. S. Jensen, C. M. Jermaine, X. Zhou (Eds.), 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, IEEE Computer Society, 2013, pp. 1081–1092. URL: https://doi.org/10.1109/ICDE.2013.6544899. doi:10.1109/ICDE.2013.6544899.

[36] X. Zhou, J. Sun, G. Li, J. Feng, Query performance prediction for concurrent queries using graph embedding, Proc. VLDB Endow. 13 (2020) 1416–1428. URL: http://www.vldb.org/pvldb/vol13/p1416-zhou.pdf. doi:10.14778/3397230.3397238.

[37] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, C. Zhang, Statistical learning techniques for costing XML queries, in: K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, B. C. Ooi (Eds.), Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, ACM, 2005, pp. 289–300. URL: http://www.vldb.org/archives/website/2005/program/paper/wed/p289-zhang.pdf.

[38] R. Marcus, O. Papaemmanouil, Plan-structured deep neural network models for query performance prediction, Proc. VLDB Endow. 12 (2019) 1733–1746. URL: http://www.vldb.org/pvldb/vol12/p1733-marcus.pdf. doi:10.14778/3342263.3342646.

[39] Y. Zhou, W. B. Croft, Query performance prediction in web search environments, in: W. Kraaij, A. P. de Vries, C. L. A. Clarke, N. Fuhr, N. Kando (Eds.), SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007, ACM, 2007, pp. 543–550. URL: https://doi.org/10.1145/1277741.1277835. doi:10.1145/1277741.1277835.

[40] J. Duggan, U. Çetintemel, O. Papaemmanouil, E. Upfal, Performance prediction for concurrent database workloads, in: T. K. Sellis, R. J. Miller, A. Kementsietsidis, Y. Velegrakis (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, ACM, 2011, pp. 337–348. URL: https://doi.org/10.1145/1989323.1989359. doi:10.1145/1989323.1989359.

[41] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, S. B. Zdonik, Learning-based query performance modeling and prediction, in: A. Kementsietsidis, M. A. V. Salles (Eds.), IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012, IEEE Computer Society, 2012, pp. 390–401. URL: https://doi.org/10.1109/ICDE.2012.64. doi:10.1109/ICDE.2012.64.

[42] C. T. Yu, M. Z. Özsoyoğlu, An algorithm for tree-query membership of a distributed query, in: The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 1979, pp. 306–312.

[43] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, D. Lemire, Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources, in: G. Das, C. M. Jermaine, P. A. Bernstein (Eds.), Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, ACM, 2018, pp. 221–230. URL: https://doi.org/10.1145/3183713.3190662. doi:10.1145/3183713.3190662.

[44] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proc. VLDB Endow. 9 (2015) 204–215. URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf. doi:10.14778/2850583.2850594.

[45] J. Leskovec, R. Sosic, SNAP: A general-purpose network analysis and graph-mining library, ACM Trans. Intell. Syst. Technol. 8 (2016) 1:1–1:20. URL: https://doi.org/10.1145/2898361. doi:10.1145/2898361.

[46] M. Lanzinger, R. Pichler, A. Selzer, Avoiding materialisation for guarded aggregate queries, CoRR abs/2406.17076 (2024). URL: https://doi.org/10.48550/arXiv.2406.17076. doi:10.48550/ARXIV.2406.17076. arXiv:2406.17076, to be presented at VLDB 2025.

[47] A. Mhedhbi, M. Lissandrini, L. Kuiper, J. Waudby, G. Szárnyas, LSQB: a large-scale subgraph query benchmark, in: V. Kalavri, N. Yakovets (Eds.), GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021, ACM, 2021, pp. 8:1–8:11. URL: https://doi.org/10.1145/3461837.3464516. doi:10.1145/3461837.3464516.

[48] D. S. Himmelstein, A. Lizee, C. Hessler, L. Brueggeman, S. L. Chen, D. Hadley, A. Green, P. Khankhanian, S. E. Baranzini, Systematic integration of biomedical knowledge prioritizes drugs for repurposing, eLife 6 (2017) 1–35. URL: https://doi.org/10.7554/eLife.26726.001. doi:10.7554/eLife.26726.001.

[49] M. Abseher, N. Musliu, S. Woltran, Improving the efficiency of dynamic programming on tree decompositions via machine learning, J. Artif. Intell. Res. 58 (2017) 829–858. URL: https://doi.org/10.1613/jair.5312. doi:10.1613/JAIR.5312.

[50] Y. Feng, H. You, Z. Zhang, R. Ji, Y. Gao, Hypergraph neural networks, in: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, AAAI Press, 2019, pp. 3558–3565. URL: https://doi.org/10.1609/aaai.v33i01.33013558. doi:10.1609/AAAI.V33I01.33013558.

[51] L. Breiman, Classification and regression trees, Routledge, 2017.

[52] D. B. G. G. M. L. D. L. C. O. R. Pichler, A. Selzer, Selective use of yannakakis' algorithm to improve query performance: Machine learning to the rescue, CoRR abs/2502.20233 (2025). URL: https://doi.org/10.48550/arXiv.2502.20233. doi:10.48550/arXiv.2502.20233. arXiv:2502.20233.