Query Answering without Join Computation: An Interactive Exploration of Practical Techniques

Matthias Lanzinger
matthias.lanzinger@tuwien.ac.at
TU Wien
Vienna, Austria

Reinhard Pichler
reinhard.pichler@tuwien.ac.at
TU Wien
Vienna, Austria

Alexander Selzer
alexander.selzer@tuwien.ac.at
TU Wien
Vienna, Austria

Abstract

The explosion of intermediate join results, frequently encountered in analytical and graph queries, remains a major performance bottleneck in modern database systems. Yannakakis' algorithm enables efficient processing of acyclic conjunctive queries by avoiding the materialisation of tuples not contributing to the final output. Despite its theoretical guarantees, Yannakakis' algorithm and its adaptations are still largely absent from modern query engines. In recent work, we have integrated Yannakakis-style query processing into Spark SQL via the extension of the query optimiser on the logical level and the introduction of a new physical operator: (Group)AggJoin.

In this demonstration, we present Spark-Y, a system for exploring and analysing Yannakakis-style query processing in Spark SQL. The system visualises query hypergraphs, join trees and execution plans, providing side-by-side performance comparisons between the standard and optimised execution. Spark-Y serves as a practical demonstration of the performance gains achievable through this approach, as well as a tool for analysing the behaviour and effects of optimisation rules in Spark SQL. Beyond performance analysis, interactive visualisations provide an intuitive understanding of how Yannakakis-style query processing works in practice.

CCS Concepts

• **Information systems → Query optimization**.

Keywords

Join Processing, Aggregate Queries, Acyclicity

ACM Reference Format:

Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. 2026. Query Answering without Join Computation: An Interactive Exploration of Practical Techniques. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31–June 5, 2026, Bengaluru, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3788853.3801583

1 Introduction

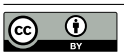
Join processing is a decisive factor in overall query performance – especially for analytical queries which join over many relations and aggregate the data to produce a compact result. The explosion of intermediate join results remains a significant challenge for relational database systems in general and specifically for analytical queries. Traditionally, query engines aim to reduce materialisation by finding a good join order. However, while this alleviates the problem, it does not eliminate the fundamental problem of producing and materialising intermediate results which do not contribute to the final output.

The algorithm by Yannakakis [4] alleviates this problem by eliminating all tuples not contributing to the final output – by reducing the relations via semi-joins and then computing the joins efficiently. This is applicable for all acyclic conjunctive queries, which covers the vast majority of queries in practice. Despite the seemingly clear advantages, Yannakakis' algorithm has not yet been adopted by production database systems. Recent work has brought renewed attention to integrating Yannakakis-style query evaluation – as opposed to strictly following Yannakakis' algorithm – into database systems [1]. Through different approaches based on this general idea, integrations into several systems (Umbra, Apache DataFusion, DuckDB, and Spark SQL) have been developed. Among these, in our previous work [2], we presented one such implementation of Yannakakis-style query evaluation specifically focused on optimising large classes of join-aggregate queries, with very promising results: significant speedups on average for challenging queries without measurable overhead on simpler queries.

In [2], we introduced the class of *piecewise-guarded aggregate queries (pw-guarded queries)*: a large subset of acyclic aggregate queries covering most queries found in benchmarks. For pw-guarded queries, we introduced logical optimisations for rewriting (sub)plans to execute a newly introduced physical operator (AggJoin) along the join tree. The AggJoin operator integrates aggregation into a semi-join-like operator, and the implementation is based on Spark SQL's hash-join and sort-merge-join operators. For pw-guarded queries, materialisation can be avoided completely due to the Yannakakis-style bottom-up traversal of AggJoins. This version of Spark SQL includes further optimisations for unguarded queries (going beyond the scope of the work in [2]) by extending the AggJoin to the GroupAggJoin operator. While materialisation cannot be fully avoided in the case of unguarded queries, preliminary results demonstrate significant performance gains for "slightly" unguarded queries.

In this demonstration, we present Spark-Y– a system for exploring, analysing, and visualising Yannakakis-style query evaluation in Spark SQL. Spark-Y was built with the goals of demonstrating the performance gains achievable in practice through this new approach for join-aggregate query processing, and providing a tool for analysing the behaviour and effects of extensions of the Spark SQL optimiser focusing on join query evaluation. Furthermore, by providing interactive visualisations, we aim to improve awareness and understanding of Yannakakis-style query processing in practice.

Spark-Y provides the following main features (among others):



SIGMOD Companion '26, Bengaluru, India

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3801583

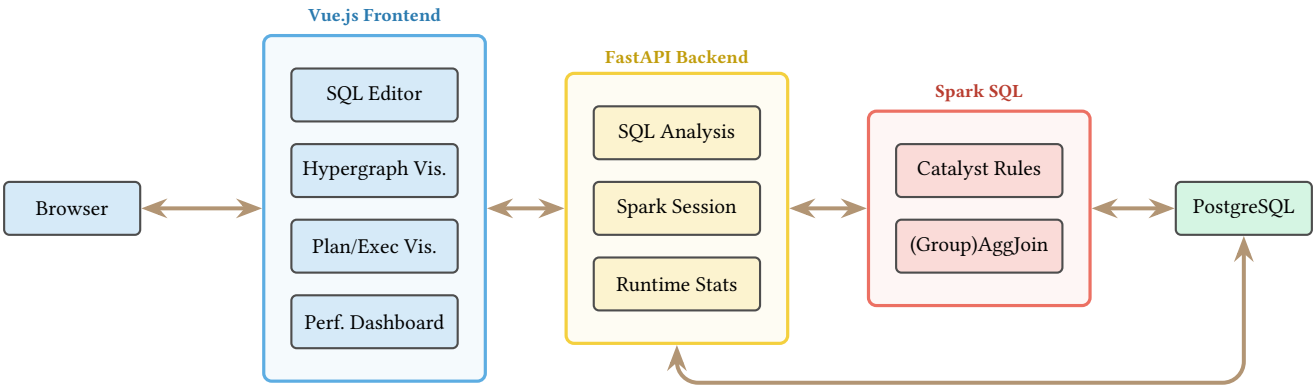


Figure 1: System Architecture

- Visualisations of query hypergraphs, the mapping of join-attribute equivalence classes in the query to hypergraph vertices, and the distribution of output attributes.
- Classification of queries (acyclic, piecewise-guarded, etc.), display of the join tree, and original and rewritten plans.
- Side-by-side performance comparison of Spark SQL with and without the optimisations.
- Visualisation of the execution DAG and physical operators of Spark SQL together with the intermediate results.
- Ready-to-go docker-compose environment including frontend, backend, Spark SQL, and a PostgreSQL database.
- Simple and quick data import of SQL dumps from any existing PostgreSQL database.

2 Fast Join-Aggregate Processing in Spark SQL

Piecewise-guarded aggregate queries. The goal of the work in [2] was to identify a large class of aggregate queries that can be evaluated by a bottom-up traversal via semi-join-like operations, without materialising any join results.

We consider queries of the following form:

$$Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big) \qquad (1)$$

where $\gamma[g_1, \ldots, g_\ell, A_1(a_1), \ldots, A_m(a_m)]$ denotes the grouping operation for attributes $g_1, \ldots, g_\ell$ and aggregate expressions $A_1(a_1), \ldots, A_m(a_m)$ for some (standard SQL) aggregate functions $A_1, \ldots, A_m$ applied to expressions $a_1, \ldots, a_m$. The grouping attributes $g_1, \ldots, g_\ell$ are attributes occurring in the relations $R_1, \ldots, R_n$ and $a_1, \ldots, a_m$ are expressions formed over the attributes from $R_1, \ldots, R_n$.

Thus, we introduced *piecewise-guarded (pw-guarded) aggregate queries*. We call a query of the form in (1) a *piecewise-guarded aggregate query* if $(R_1 \bowtie \cdots \bowtie R_n)$ is acyclic and there exists a relation R_i (= the *guard*) that contains all attributes that are part of the grouping clause or any unguarded aggregate expressions (where for the most common aggregate functions SUM, COUNT, MIN, MAX, AVG, expressions may have their own guard).

This *piecewise-guardedness* condition covers the vast majority of aggregate queries found in practice, including most queries in standard benchmarks. Indeed, in the JOB, STATS-CEB, and SNAP benchmarks, 100% of the queries are aggregate-join queries or contain aggregate-join subqueries, and all are piecewise-guarded. In other benchmarks such as TPC-H, TPC-DS, and LSQB, between half and two thirds are (or contain) aggregate-join (sub)queries and approximately 50% of them are piecewise-guarded.

Integration into Spark SQL. The optimisations are realised as logical (plan-level) and physical optimisations (physical operators). In newly introduced Spark SQL optimiser rules, pw-guarded queries in sub-plans are matched, and their join core is converted to a hypergraph, from which a join tree is constructed (if the query is acyclic). The sub-plan is rewritten to follow the join tree's structure.

For the physical execution, we introduced the *AggJoin* operator, which integrates aggregation directly into a semi-join-like operator. Rather than first computing joins and then aggregating, AggJoin computes partial aggregates during the traversal along the join tree, eliminating intermediate result materialisation entirely for pw-guarded queries. For queries that are acyclic but not pw-guarded (*unguarded queries*), we extended the AggJoin operator to the *GroupAggJoin*. This operator handles cases where some output attributes lack guardedness by performing grouped aggregation, reducing, while not fully eliminating, intermediate result materialisation.

Performance. Experimental evaluation on several standard benchmarks confirms that the optimisations yield substantial improvements in performance. In the following table, the end-to-end runtimes for 4 benchmarks are shown, with *Ref* referring to the original runtime and *Opt* to the runtime with the optimisations enabled:

Query/Queries	Ref	Opt	Speedup
STATS-CEB e2e	1558±7.3	**64.8**±7.9	24.04 x
JOB e2e	3217.84±106	**2189.46**±76	1.47 x
TPC-H e2e SF200	3757.2±75	**3491.06**±27	1.08 x
LSQB Q1 SF300	3096±232	**688**±23	4.57 x
LSQB Q4 SF300	602±37	**592**±9	1.02x

For queries on graph data, even more significant speedups can be achieved. While Spark SQL quickly runs out of memory on increasing-length path queries, with the optimisations enabled these do not pose a problem. For example, on the SNAP Google graph, a path query with 5 joins results in an error for standard Spark SQL, while running in ≈ 8 seconds with the optimisations [2].

3 Introducing Spark-Y

Spark-Y consists of two main components: a frontend application providing a web interface to the user, and a backend application for the interaction with Spark SQL and the database. A diagram of the system's architecture is shown in Figure 1.

Backend. The backend application manages the connections to both the PostgreSQL database and Spark SQL (specifically, PySpark). It handles the analysis of SQL queries (classification as guarded/piecewise-guarded, or not, as well as acyclicity checks) by making use of Spark SQL's parser, and analysing the logical plan generated. Upon user request, the backend executes queries in Spark SQL, with optimisations enabled as well as disabled, and gathers execution statistics from the Spark API.

Frontend. The frontend application provides an interactive graphical interface to the user, communicating with the backend via a REST API. The frontend consists of 4 main views: *Data Import*, *Query Explorer*, *Settings*, *Execution & Analysis*.

Data Import: The user can enter connection details for a PostgreSQL database or use the default database provided with the docker-compose environment. Data from the connected PostgreSQL database can be loaded into Spark SQL as temporary tables. To allow convenient imports of data, SQL dump files can also be inserted directly into the database from the frontend or by selecting them from the list of pre-existing dump files. The Data Import view also provides a feature for quickly sampling from a database to create a reduced version of it, saving it into the SQL dumps library – which makes it easy to create reduced databases from large fixed-size benchmarks (such as JOB [3]). One such sampled version of JOB is included in the git repository to allow users to get started with the system straight-away.

Query Explorer: The backend application contains a directory structure of benchmarks (or simply groups of queries) and queries. We have provided all queries from the JOB benchmark here.

Settings: In the Settings view, users can configure how Spark SQL executes the queries and applies optimiser rules by setting custom configuration options. By default, the "standard" version of Spark SQL is compared to the version with the AggJoin and logical optimisations enabled.

Execution & Analysis: The core interface of Spark-Y provides a SQL query editor, and the option to analyse the query or to run it as well using both implementations. The query-hypergraph is displayed using a bubble set visualisation or a visualisation based on pie charts. Equivalence classes of the SQL query corresponding to the attributes of the hypergraph are highlighted upon hovering over the query, as well as the corresponding vertices in the hypergraph. Output attributes are colored, providing an overview at a glance. Their distribution is relevant for classifying a query as piecewise-guarded, or, in the case of unguarded queries, for determining the amount of materialisation that cannot be avoided. A query is classified as cyclic or acyclic, and if acyclic as guarded, piecewise-guarded, or unguarded. In case the query is acyclic, a join tree is displayed. The logical plans of both executions can be compared in a side-by-side view. Besides the time breakdown of both systems over planning and execution runtime, detailed metrics, such as the size of shuffle reads and the total number of input rows are shown. The physical execution of both queries is displayed as a DAG, with the edges annotated by the sizes of intermediate results. The Spark-Y system is provided here: https://github.com/dbai-tuw/Spark-Y.

4 Demonstration

During the demonstration, the audience will be able to interact with Spark-Y, and experiment with a set of provided queries (we will provide at least the JOB benchmark and the corresponding queries) or write their own. We now walk through an interaction with Spark-Y– ***Run JOB query 33A and analyse the original and optimised executions***.

1. Data Import and Query Explorer. The first step involves loading the data from the JOB benchmark (the IMDB dataset) into Spark SQL. We have made it simple to get started by loading the reduced SQL dump provided in the repository in the Data Import view, then loading the data from the PostgreSQL database into Spark SQL.

In the query explorer, all queries of the JOB benchmark can be found, including JOB query 33A. For each query, the number of joins and aggregates is shown. Query 33A joins 8 tables and performs 6 MIN-aggregates over 6 attributes in distinct relations.

2. Query to Hypergraph Mapping. Query 33A can be analysed to visualise the hypergraph, or evaluated directly.

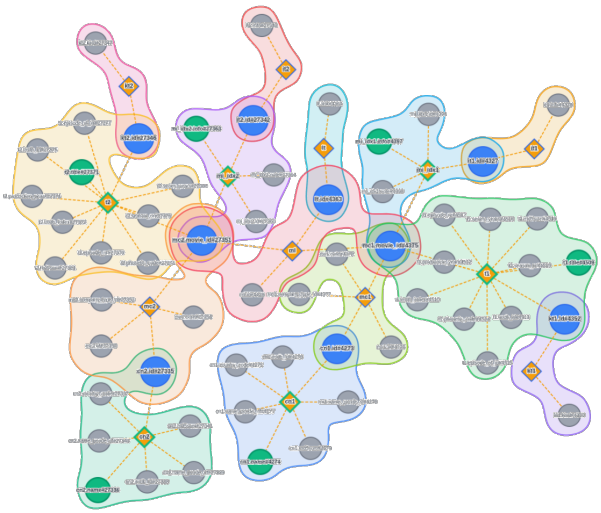


Figure 2: Hypergraph visualisation of query 33A

The hypergraph visualisation supports multiple modes, such as the bubble sets representation (vertices are covered by colored bubbles representing hyperedges), as in Figure 2. Vertices corresponding to join attributes are blue, those corresponding to output attributes are green, and those corresponding to other schema attributes are gray. In this query, we can observe that the green output attributes are spread among distinct relations. Hence, the query cannot be guarded, but due to only having MIN-aggregations, it is piecewise-guarded.

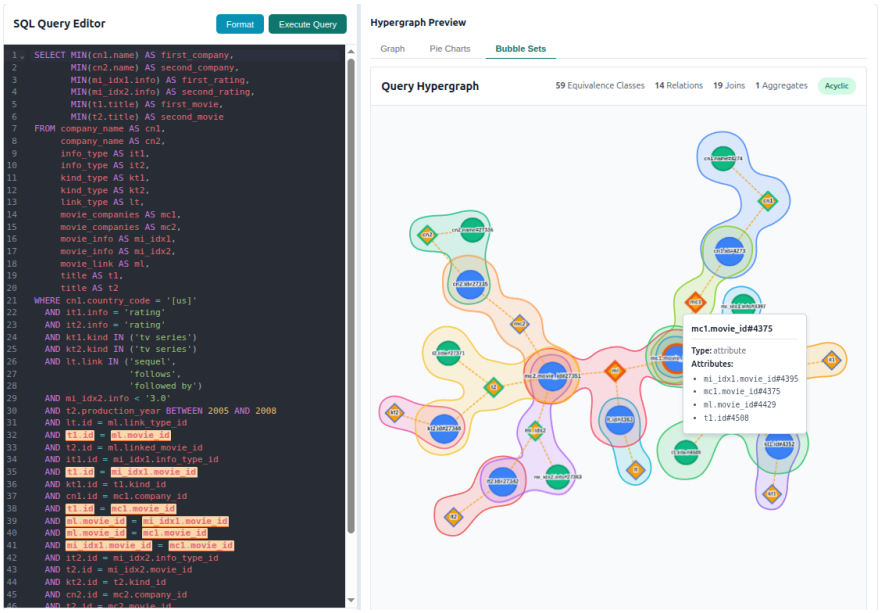


Figure 3: Query and hypergraph side-by-side

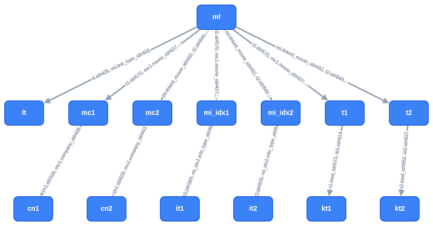


Figure 4: Join tree of query 33A

As shown in Figure 3, the query editor and the hypergraph visualisation are linked by highlighting their corresponding parts: equivalence classes of query attributes to hypergraph vertices. Hovering over an attribute in the query highlights the corresponding vertex in the hypergraph, and vice versa.

3. Join Trees and Logical Plans. Queries are checked for acyclicity. In case of an acyclic query, a join tree is displayed. Figure 4 shows the join tree displayed for query 33A.

Now, we run query 33A. Spark-Y executes the query twice: once with the optimisations enabled and once with them disabled.

4. Runtime Metrics. Spark-Y collects detailed runtime and execution metrics from Spark SQL. Figure 5 compares the runtime breakdown into planning and execution time of the two systems.

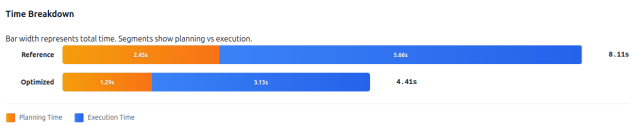


Figure 5: Runtime comparison

More detailed metrics on shuffle read/write sizes and the total number of rows produced are also provided in the frontend in a tabular overview.

5. Comparing Physical Plans. Spark-Y also compares the final physical plans as reported by the Spark API. Our aim here was to provide a clearer alternative to the Spark Web UI execution DAG. This makes it possible to hide node types (such as Sort, Shuffle, and Projection) and integrate the intermediate result sizes directly into the graph visualisation. Figure 6 shows the original Spark SQL physical plan (left) compared to the optimised plan (right).

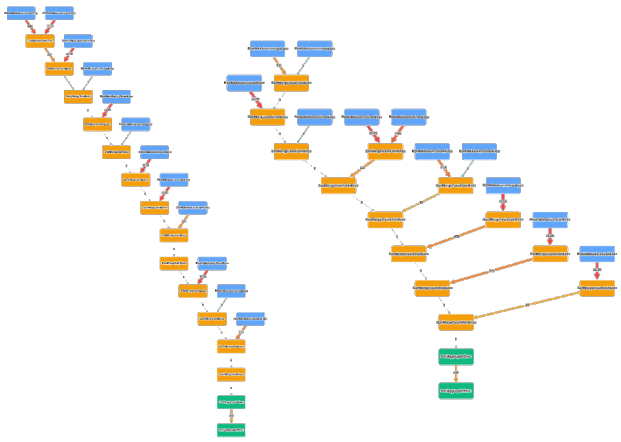


Figure 6: Physical operator DAGs

Acknowledgments

This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013].

References

[1] Paraschos Koutris, Stijn Vansummeren, Qichen Wang, Yisu Remy Wang, and Xiangyao Yu. 2026. Database Theory in Action: Yannakakis' Algorithm. In *29th International Conference on Database Theory, ICDT 2026, Tampere, Finland, March 24-27, 2026 (LIPIcs)*. Schloss Dagstuhl, 25:1–25:6. doi:10.4230/LIPICS.ICDT.2026.25

[2] Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. 2025. Avoiding Materialisation for Guarded Aggregate Queries. *Proc. VLDB Endow.* 18, 5 (2025), 1398–1411. https://www.vldb.org/pvldb/vol18/p1398-selzer.pdf

[3] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. doi:10.14778/2850583.2850594

[4] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings VLDB*. VLDB, 82–94.