Analysis of Order of Joins in Yannakakis Algorithm.

MATHESH THIRUMALAI, University Of California, Santa Cruz, USA
SAMIKSHA VARPE, University Of California, Santa Cruz, USA

Abstract - This research explores the implementation of the Yannakakis algorithm in Python, evaluating its efficiency and effectiveness through rigorous testing using the Join Order Benchmark (JOB) queries on the IMDB dataset. The Yannakakis algorithm is well-regarded for its use of semi-joins to eliminate dangling tuples prior to the join phase, thereby optimizing join operations. While it is considered an effective approach for relational joins, the impact of the order of semi-joins on the algorithm's performance remains an underexplored area. In this study, we systematically investigate the effect of join order on the algorithm's efficiency and accuracy. Our findings reveal that the order of semi-joins significantly influences the overall performance of the Yannakakis algorithm, underscoring the importance of optimizing join sequences for practical database applications.

ACM Reference Format:
Mathesh Thirumalai and Samiksha Varpe. 2024. Analysis of Order of Joins in Yannakakis Algorithm.. 1, 1 (December 2024), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Yannakakis's algorithm [1] is a foundational query optimization technique designed for efficiently evaluating acyclic join queries in relational databases. Renowned for its use of semi-joins to minimize intermediate results, the algorithm has proven to be theoretically optimal for certain classes of queries. However, despite its advantages, real-world adoption of Yannakakis's algorithm remains limited. For instance, systems like PostgreSQL rely heavily on cost-based query planners incorporating histograms, multivariate statistics to optimize join orders and reduce computational costs. Various factors contribute to the limited adoption of Yannakakis's algorithm in real-world systems, and one significant factor is the ambiguity in determining the optimal order of joins. While the algorithm efficiently reduces dangling tuples, it does not prescribe a specific sequence for performing the semi-joins or final joins. This raises the question: Does join order significantly affect the algorithm's performance? To address this, we systematically evaluated the algorithm's performance using the Join Order Benchmark (JOB) dataset. For each query in the dataset, we explored all possible join order permutations and measured the execution times. The results were striking, certain join orders outperformed others by a factor of five, highlighting the substantial impact of join order on query performance. These findings underscore the importance of join order optimization and point to potential enhancements in the practical deployment of Yannakakis's algorithm.

2 Technical Sections

2.1 Yannakakis's algorithm

The Yannakakis algorithm was introduced by Mihalis Yannakakis in 1981 as part of his research into optimizing acyclic query evaluation. It became a cornerstone in database query optimization due to its structured two-phase approach, significantly reducing intermediate results and offering efficient solutions for evaluating complex relational queries.

Authors' Contact Information: Mathesh Thirumalai, University Of California, Santa Cruz, USA, mthiruma@ucsc.edu; Samiksha Varpe, University Of California, Santa Cruz, USA, svarpe@ucsc.edu.

2024. ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Yannakakis's algorithm is an optimization technique for evaluating acyclic joins in relational databases. Its two-phase approach includes:

Full Reducer Phase

- `Bottom-Up Semi-Join:` : Starts at the leaf nodes of the join tree, eliminating tuples that cannot contribute to the final result.
- `Top-Down Semi-Join:` : Propagates constraints from the root to the leaves, further refining relations.

Join Phase

- `Join Phase`: Reduced relations are joined sequentially, following the join tree structure, to compute the final query result efficiently.

By eliminating irrelevant tuples during the Full Reducer phase, the algorithm minimizes intermediate results and reduces computational overhead. Unlike traditional methods that perform joins first and filter later, Yannakakis's method avoids unnecessary Cartesian products, making it highly efficient for acyclic queries.

2.2 Job Dataset

The Join Order Benchmark (JOB) [2] dataset is a set of realistic, analytical SQL queries based on the Internet Movie Database (IMDB). It is designed to test query optimization, specifically focusing on join order selection, which is crucial for database performance. The dataset includes 21 tables with rich inter-table relationships and significant correlations, challenging cardinality estimation algorithms. Queries in JOB average 8 joins per query, providing a diverse and complex workload to evaluate optimizers. JOB emphasizes real-world constraints, highlighting practical optimization challenges.

Using the JOB dataset with Yannakakis's algorithm is particularly insightful because the algorithm's effectiveness hinges on semi-join and join order efficiency. By leveraging JOB, we can measure how varying join orders influence performance and evaluate the algorithm's ability to optimize and handle large-scale realistic workloads.

2.3 Order of Joins

To evaluate the impact of different join orders, we implemented a function capable of generating all possible permutations of join sequences in the Yannakakis algorithm. For example, a query from the JOB dataset involving five joins results in 5! permutations, amounting to 120 possible join orders. Each permutation was executed using the Yannakakis algorithm, and the corresponding execution times were meticulously recorded for analysis.

2.4 Performance Estimation

To evaluate the performance of various join order in the Yannakakis algorithm in Python, we adopt a structured comparison approach focusing on both time and space metrics:

- **`Time Measurement`**: For the Python implementation, the time taken for each phase of Yannakakis like Bottom-Up Semi-Join, Top-Down Semi-Join, and Join Phase, is logged using Python's time module.
- **`Space Measurement`**: In Python, memory usage is tracked after selections, semi-joins, joins, and projections using sys.getsizeof.
- **`Dangling Tuples`**: To gain an additional perspective on performance, we also measured the volume of dangling tuples eliminated during each phase of the full reducer process, encompassing both the bottom-up and top-down semi-join reductions.

3 Implementation

Below depicts the pesudocode of the Yannakakis algorithm being implemented :-

```
Function Yannakakis(relations, join_combinations,
    selection_criteria):
     For each join_tree in join_combinations:
         # Apply Selections:
             For each relation in selection_criteria:
                 Filter rows based on conditions using
                     apply_selection().

         # Bottom-Up Semi-Join Phase:
             For each edge in reversed join_tree:
                 Perform semi-join to filter tuples in parent
                     relation.

         # Top-Down Semi-Join Phase:
             For each edge in join_tree:
                 Perform semi-join to filter tuples in child
                     relation.

         # Join Phase:
             Initialize result as the reduced form of the root
                 relation.
             For each edge in join_tree:
                 Perform join between relations using join().

         # Return Final Result.
```

4 Experiments

4.1 Setup Details

Following is technical specifcation of the setup used :-

- **Disk Size:** 950 GB
- **Operating System:** Windows 11
- **Processor:** Intel(R) Core(TM) Ultra 9 185H 2.30 GHz
- **RAM:** 32 GB
- **Python Version:** 3.12
- **Pandas Version:** 2.2.3

4.2 Analysis of Join Order Impact on Algorithm Performance

To ensure the correctness of the Yannakakis algorithm, it was initially benchmarked against PostgreSQL for validation. The join order was preserved as provided in the input, with no optimizations or reordering applied. Additionally, projections were omitted to simplify the implementation, as they do not directly impact the evaluation of the algorithm itself. The results, depicted in Figure 1, indicate that the algorithm performs comparably to PostgreSQL, demonstrating similar execution times. This alignment validates the accuracy and consistency of the Yannakakis algorithm in handling the tested query scenarios.

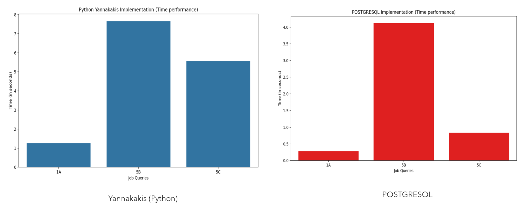


Fig. 1. Performance comparison (Python (Left) Vs PostgreSQL(Right)

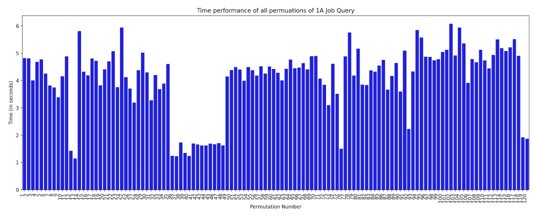


Fig. 2. All combinations of 1A Job Joins

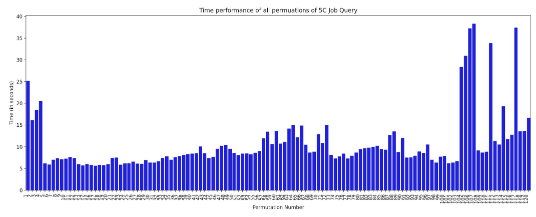


Fig. 3. All combinations of 5C Job Joins

After successfully validating the algorithm's correctness, comprehensive tests were conducted using all possible join order permutations. Due to computational constraints, JOB queries 1A and 5C were selected for this analysis. Both queries involve 5 joins, resulting in 5! = 120 distinct join order combinations. Each combination was executed using the Yannakakis algorithm, and the corresponding execution times were recorded. The results are visualized in Figures 2 and 3, representing the performance outcomes for JOB queries 1A and 5C, respectively. These results provide valuable insights into the impact of join order permutations on query execution efficiency.

The results reveal striking differences in query execution times for JOB queries 1A and 5C, highlighting the significant impact of join order. For query 1A, the best-performing permutation achieved an execution time of **1.14 seconds**, while the worst-performing permutation required **6.07 seconds**, a nearly fivefold difference. Similarly, for query 5C, the best-performing permutation completed in **5.6 seconds**, whereas the worst-performing permutation took **38.3 seconds**, exhibiting a sixfold disparity. These findings are surprising, because despite employing the same Yannakakis algorithm, the execution time can vary drastically based solely on the order of joins. This underscores the critical importance of optimizing join order for efficient query execution.

To investigate the significant performance differences observed in the join order experiments, we conducted an in-depth analysis by dissecting the queries and examining their underlying data distributions. The results indicate that selection operations have a substantial impact, particularly by creating a high number of dangling tuples and inducing significant data skew. The issue becomes evident when considering the distribution of movie_info_idx.info_type_id and info_type.id, as depicted in Figures 4 and 5. Despite these columns being unequally distributed, their direct join would not have caused a considerable bottleneck. However, the application of selection criteria to info_type.id, as shown in Figure 6, drastically reduces the result set to a single row (Figure 7). This sparsity introduces a substantial number of dangling tuples in the movie_info_idx table, leading to inefficiencies during join processing. A similar scenario is observed with the columns company_type.id and movie_companies.company_type_id. The initial distribution of movie_companies.company_type_id contains approximately 2.3 million rows. However, post-selection, this number plummets to 17,500 rows, leaving nearly 2 million dangling tuples in the movie_companies table. This dramatic reduction highlights the impact of skewed selections on data distributions, which exacerbates the inefficiency in processing. Skewed distributions and sparse selections not only increase the number of dangling tuples but also amplify the computational burden during the semi-join phases.

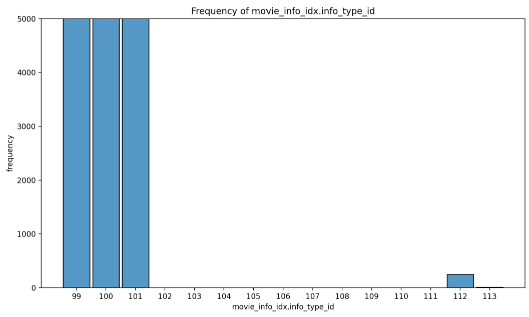


Fig. 4. Frequency of movie_info_idx.info_type_id (Values are scaled down to cover entire distribution).

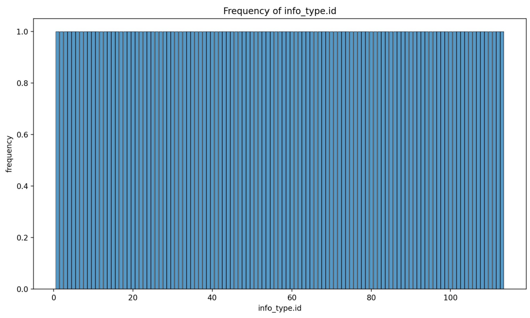
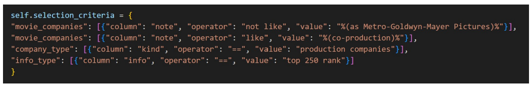
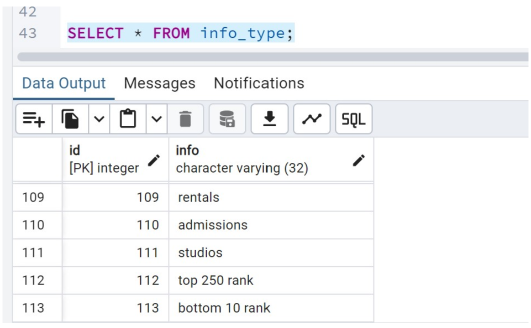


Fig. 5. Frequency of info_type.id. (Here all the values are just 1)



```
self.selection_criteria = {
"movie_companies": [{"column": "note", "operator": "not like", "value": "%(as Metro-Goldwyn-Mayer Pictures)%"}],
"movie_companies": [{"column": "note", "operator": "like", "value": "%(co-production)%"}],
"company_type": [{"column": "kind", "operator": "==", "value": "production companies"}],
"info_type": [{"column": "info", "operator": "==", "value": "top 250 rank"}]
}
```

Fig. 6. Selection criteria of JOB Dataset 1A.


```
SELECT * FROM info_type;
```

	id [PK] integer	info character varying (32)
109	109	rentals
110	110	admissions
111	111	studios
112	112	top 250 rank
113	113	bottom 10 rank

Fig. 7. Table view of info_type.id table. There is just one top 250 rank row which is 112.

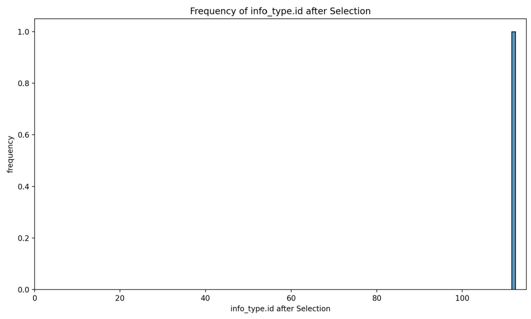


Fig. 8. Histplot of info_type.id after selection. Only 112 remains

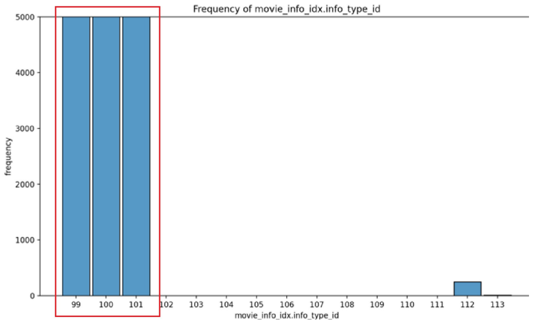


Fig. 9. Histplot of movie_info_idx.info_type_id before selection. The highlighted values all become dangling tuples.

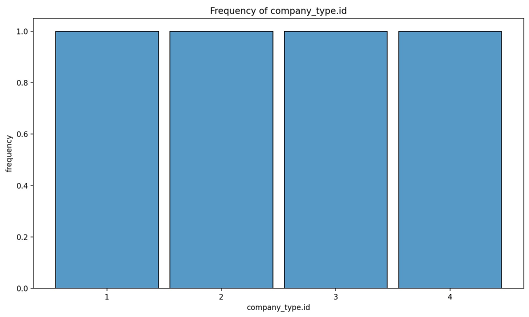


Fig. 10. Histplot of company_type.id. All values are just 1.

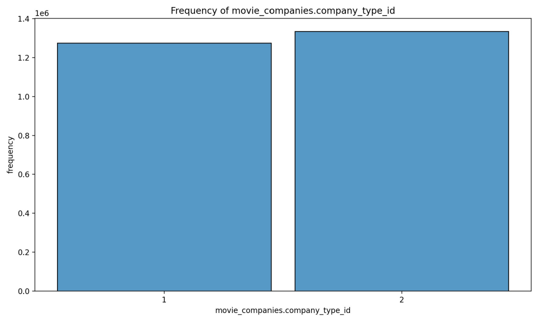


Fig. 11. Histplot of movie_companies.company_type.id. Values are in range 1.2 - 1.3 Million

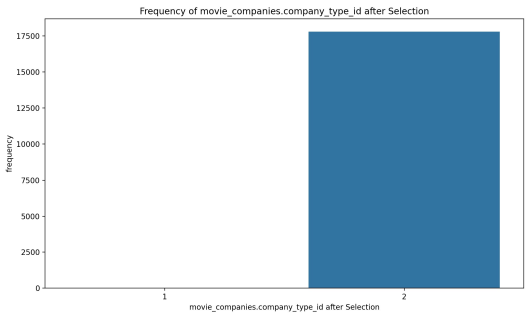


Fig. 12. Histplot of movie_companies.company_type.id after selection. Just 17.5K remain

Examining both relations, it is apparent that prioritizing the second relation, which has a higher number of dangling tuples, might initially seem beneficial. However, closer analysis reveals that the join between movie_info_idx and info_type effectively eliminates approximately 1.3 million tuples. This significant reduction minimizes the volume of unwanted tuples carried into subsequent relations, demonstrating that the sequence of joins can profoundly influence performance. Our findings indicate that it is not merely the volume of dangling tuples in isolation that impacts efficiency but also the interdependence among joins. In this example, completing the joins between movie_info_idx and info_type, as well as between title and movie_info_idx, at an early stage reduces the proportion of dangling tuples to less than 1%. A key observation is that once dangling tuples are minimized in one relation, subsequent joins involving that relation effectively eliminate further dangling tuples. This cascading reduction underscores the importance of strategically ordering semi-joins within the full reducer phase to optimize performance. The list below outlines the number of dangling tuples removed at each step of the full reducer phase, illustrating the cumulative effect of the join order on efficiency.

- Number of Dangling tuples removed in Bottom-up Semi-Join phase:
  - Approximately 1.37 million tuples from movie_info_idx and info_type join
  - Around 17,793 tuples from movie_companies and movie_info_idx join
  - Roughly 2.52 million tuples from title and movie_info_idx join
  - Zero tuples from company_type and movie_companies join
  - Zero tuples from movie_companies and title join
- Number of Dangling tuples removed in Top-Down Semi-Join phase:
  - 242 tuples removed from movie_companies and title join
  - Zero tuples from company_type and movie_companies join
  - 242 tuples from title and movie_info_idx join
  - Zero tuples from movie_companies and movie_info_idx join
  - Zero tuples from movie_info_idx and info_type join

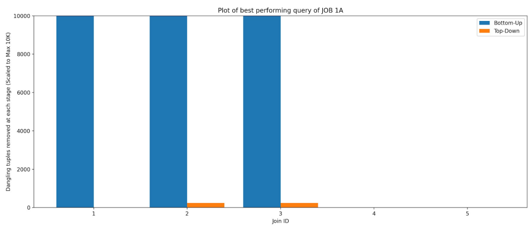


Fig. 13. Bar Plot of best performing queries, here the max limit is set at 10K, to make sure lower limit is visible. (Max was 2 million)

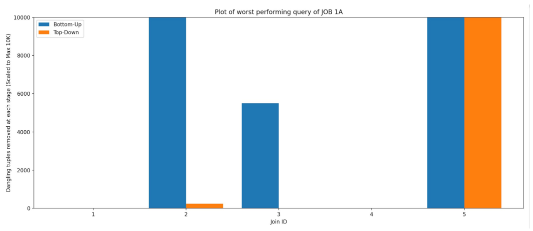


Fig. 14. Bar Plot of worst performing queries, here the max limit is set at 10K, to make sure lower limit is visible. (Max was 2 million). Some top-down queries has 5 dangling tuples which has not shown.

Figures 13 and 14 depict the reduction in dangling tuples during the full reducer phase for the best-performing and worst-performing queries respectively of the 1A query from the JOB dataset. The results clearly demonstrate that the superior performance in the best query is attributed to the effective removal of dangling tuples within the first three iterations of the bottom-up semi-join phase. This early elimination significantly reduces the number of tuples carried forward into subsequent phases, including the top-down semi-join, thereby minimizing computational overhead in later stages.

In contrast, Figure 14 highlights a suboptimal scenario where the majority of dangling tuples are only removed during the final iteration of the semi-join phase. This delay causes a substantial increase in the number of tuples processed in earlier iterations, leading to an inflated number of operations. While both cases exhibit linear time complexity in practice, the worst-performing query suffers from a significantly higher constant factor due to the excessive number of tuples involved at each step. This stark difference underscores the critical impact of join order on the efficiency of the Yannakakis algorithm.

5 Future Work

In future work, our primary focus will be on exploring the intricate relationships between join operations and their impact on the overall performance of join order execution. A deeper understanding of these relationships will enable us to develop more efficient strategies for join ordering within the Yannakakis algorithm. We also aim to integrate a robust cardinality estimation method into our framework to optimize join order decisions. Specifically, this step would be placed immediately after the selection phase and prior to the full reducer phase.

Inspired by the sophisticated approach introduced by Woltmann et al. [3], we are particularly interested in leveraging localized deep learning models for cardinality estimation. Their method, which decomposes the database schema into smaller, specialized relations, a concept referred to as the "local approach" offers a promising avenue for achieving more accurate predictions of result sizes. By fine-tuning model parameters for these localized subsets, their framework demonstrates significant potential in overcoming the challenges posed by diverse data distributions and query patterns.

We find this approach highly promising and compelling because data characteristics vary widely, making it nearly impossible to define a universal cardinality estimation function. Real-world data exhibits varying distributions and complexities, rendering a universal solution nearly unattainable. However, breaking down the problem into smaller, manageable components and building a specialized framework, as demonstrated in the referenced paper, provides a pragmatic and scalable path forward. This localized methodology not only addresses the challenges of data heterogeneity but also lays the groundwork for crafting tailored cardinality estimation strategies that can adapt to diverse contexts effectively.

6 Conclusion

The Yannakakis algorithm, despite having its theoretical time complexity of $O(n + |OUT|)$, demonstrates a significant dependency on the order of joins, as evidenced by performance variations exceeding a factor of 5X in our study. This underscores a critical challenge in practical applications and may partly explain the algorithm's limited adoption in industrial settings. Our investigation revealed that optimal join order is not always dictated by the number of dangling tuples but rather by the intricate relationships between tables. This highlights the necessity of advanced cardinality estimation techniques to determine efficient join orders.

However, designing a universal cardinality estimator remains a formidable challenge due to the diverse nature of datasets. We propose that rather than seeking a single solution, a composite framework employing deep learning-based models tailored to specific data characteristics could offer a promising pathway. This multifaceted approach could address current limitations, paving the way for more robust and adaptive implementations of the Yannakakis algorithm in real-world scenarios.

7 Addressing Feedback

- Below are the feedbacks which are addressed from the intermediate project report :-
  - Removed Abstract Details like controlled lab environment.
  - Enhanced the abstract and introductory sections to define the research goals explicitly, emphasizing what the study seeks to achieve and the practical implications of the findings.
  - Revised the document to eliminate inconsistencies in tense, ensuring a consistent and formal tone throughout.
  - Updated all references to align with academic conventions by using authors' last names, providing clear attribution.
  - Expanded the discussion on the integration of cardinality estimation into the proposed approach, offering precise details on its intended application and expected impact.

References

[1] M. Yannakakis, "Algorithms for acyclic datarase schemes," *Proceedings of the seventh international conference on Very Large Data Bases*, vol. 7, 1981.

[2] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, pp. 204–215, 2015.

[3] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner, "Cardinality estimation with local deep learning models," *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pp. 1–8, 2019.