Reviewer(s)' Comments to Author:
Referee: 1
Recommendation: Needs Minor Revision
Comments:
The paper presents an interesting variant for linear-time evaluation of acyclic queries. Instead of using the somewhat expensive Yannakakis preprocessing strategy, the elimination of non-matching tuples is piggy-backed into regular join processing: When a tuple finds no join partners, it is eliminated from its owning hash table, thus never causing additional lookups again. That is a neat idea that creates no additional costs for joining tuples and only a little overhead for non-joining tuples, which makes the semi-join reduction more a less free.

In a practical implementation there might be some issues with that strategy. How can we safely eliminate tuples during multi-threaded execution? All experiments were performed single threaded, which avoids this problem, but in reality one would want to use multi-threaded execution, which makes elimination more tricky. It is still certainly doable, but it will be more complex and potentially also a bit more expensive, as tuple elimination will require synchronization primitives or atomic operations, which increase the overhead.

Another aspect that the authors should discuss at least briefly is the handling of cartesian products and non-equality joins. The papers mentions that these are ignored for simplicity, but that is not a viable strategy if a query includes one of these. Admittedly these queries are uncommon, but they do exist, and a system has to do something about them. At least a few sentences on how these cases can be handled would be good, even if the handling will be sub-optimal.

In Section 5, the purpose of the No-Good List remains a bit unclear to me. The authors suggest to put values from the start relation that do not find join partners into a dedicated hash table, to stop pruning early. But why is that necessary at all? Assume our relations are S(x), R(x,y), T(y,...) etc. For a given x value, we will first probe R, and then, if that does not lead to further join partners in T, eliminate the corresponding tuples from R. Thus, after the second lookup for a failed x value will always return an empty list, which immediately stops joining. Why would a No-Good List help in this case? It would have the same effect, but it would cost another hash table lookup (and another hash table, which can be non-negligible if many values find not join partner). Perhaps I misunderstood the explanation of that mechanism, but currently it does not seem to be necessary.

The related work section should discuss the pros and cons of the proposed approach relative to this paper:

Yisu Remy Wang, Max Willsey, Dan Suciu: Free Join: Unifying Worst-Case Optimal and Traditional Joins. Proc. ACM Manag. Data 1(2): 150:1-150:23 (2023)

Overall, while the strategy proposed in this paper is more a small twist to regular join processing than a huge new idea, it is a nice improvement that with relatively little effort can greatly speed up join processing, at least for some queries. What is also nice about this paper that is that it discusses cyclic queries, too, which again are more uncommon but which have to be handled by full-fledged systems.

Additional Questions:
Relevance to Databases: High

Significance of Contribution: Marginal

Readability and Organization: High

Fusion of Theory and Practice: Adequate

Length (Relative to the useful contents of the paper): Just Right

Please help ACM create a more efficient time-to-publication process: Using your best judgment, what amount of copy editing do you think this paper needs?: Light

Most ACM journal papers are researcher-oriented. Is this paper of potential interest to developers and engineers?: Maybe

Referee: 2
Recommendation: Needs Major Revision
Comments:
The paper proposed another Yannakakis-style join algorithm by showing the connection between Constraint Satisfaction Problems and the Conjunctive Query Evaluation Problem. The theoretical analysis and experimental evaluation have shown that the new method can achieve theoretically optimal O(IN+OUT) time for acyclic queries, and can further extend to support cyclic queries, while significantly reducing the hidden constant cost from the Yannakakis Algorithm. The paper presents some interesting results, which may motivate further research and potential implementations in industrial

products. However, a lot of details are missing from the papers. Thus, I believe a major revision is needed to accept this Paper.

Detailed Comments:

1. As the authors' suggestion, the Yannakakis algorithm is difficult to integrate into existing systems. However, the proposed algorithm has not been implemented in any database systems either. As far as I can see, the algorithm uses a multi-way (pipelining) hash join, with additional backjumping and tuple removal, which still requires implementing new join operations in current database systems. On the other hand, the recent work Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees, SIGMOD 25 have shown that the Yannakakis algorithm can be implemented in any database system without modifying the database itself. The authors should provide some justifications for the ease of implementation of the TreeTracker Join algorithm in the current database system.

2. The algorithm design of the TreeTracker Join algorithm relies on pipelining, which means the join order used by TreeTracker Join must be a left-deep plan. However, for the standard Yannakakis algorithm, a bushy plan with materialization can also be an option. After two rounds of semi-joins, the Yannakakis algorithm can evaluate the full join using any join order. The authors should explain why left-deep is always beneficial here.

3. I also realize that, despite being invented from different aspects, the proposed algorithms are almost identical to some recent works on dynamic query evaluation, such as Dynamic Yannakakis or CROWN. For example, in CROWN, if we consider the entire database as an insertion-only update sequence, and we initial the hash-based data structure by inserting the database following a given join tree structure in a bottom-up fashion, i.e., we only start the insertion of a relation R if all tuples for its child nodes are inserted. Such a cost will be the same as the cost of building all hash tables in the TreeTracker Join, and during this procedure, it already performs tuple removal. It won't need to do backjumping as the algorithm can guarantee that no failure exists. After initializing the data structure, we can use it to evaluate the full join query in a similar fashion to TreeTracker Join.

The authors should cite and compare these works and highlight the differences.

The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates, SIGMOD 17
Change Propagation without Joins, VLDB 23

4. The new concept tree convolution is interesting, and I will consider that the most important contribution of this paper. However, some concepts are not well explained. For example, nested convolution is not defined. I suggest that the authors polish these results with a formal definition and better-motivated examples. In addition, I would like to see a detailed comparison between Hypertree Decomposition and Tree Convolution. For example, we can also avoid materialization if the given hypertree decomposition contains only one bag with multiple relations, by rotating the hypertree to use that bag as the root node, and evaluating the given query using any left-deep plan starting from the root node (the bag of relations). The resulting query plan can also be pipelined and without materialization.

5. Recently, a lot of efforts have been made to adopt Yannakakis-style algorithms inside database systems, including
Debunking the Myth of Join Ordering: Toward Robust SQL Analytics, SIGMOD 25
Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees, SIGMOD 25
Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries, CIDR 23
The authors should discuss the difference between these works and TTJ, and compare TTJ's performance with this work. In addition, it would be better for the authors to use a fully in-memory database like DuckDB as the baseline to justify the new algorithm's efficiency better.

Additional Questions:

Relevance to Databases: Very High

Significance of Contribution: Marginal

Readability and Organization: Marginal

Fusion of Theory and Practice: Adequate

Length (Relative to the useful contents of the paper): Too Short

Please help ACM create a more efficient time-to-publication process: Using your best judgment, what amount of copy editing do you think this paper needs?: Moderate

Most ACM journal papers are researcher-oriented. Is this paper of potential interest to developers and engineers?: Maybe

Referee: 3

Recommendation: Needs Major Revision

Comments:
== Paper Contribution Summary
The paper proposes an algorithm for (full) acyclic join queries that achieves linear time in the input and output size, as the classic Yannakakis algorithm does. While Yannakakis semi-joins all relations up-front, bottom-up, the proposed algorithm (TTJ) deletes tuples on-the-fly, as the relations are traversed top-down. This is not a surprising development, as Yannakakis is known to be dynamic programming, and the top-down evalutation is a standard strategy for dynamic programs. TTJ can also be considered an adaptation of TreeTracker, an algorithm for CSP problems. The authors show that on any given query plan, TTJ will outperform the standard binary Hash-join algorithm, at least in terms of hash lookups. The experimental evalutation compares TTJ mainly with Hash-join and Yannakakis and shows that TTJ usually works better in practice. Finally, the authors show how the algorithm applies to cyclic queries, by developing the notion of tree convolution, which can be thought of as a nested join tree.

== Strong Points

S1. The delevopment of a performant Yannakakis-like algorithm in standard query execution engines is an important problem, given the prevalence of this algorithm in the theory community. The paper partially achieves this, besides the fact that the implementation is not integrated with any DB system.

S2. The paper achieves a very nice balance of theory and practice. In particular, it establishes the connection between DB theory concepts (join trees) and the corresponding DB system concepts (query plans) in a way that is novel, at least to me.

S3. The experimental results are promising and indeed show that TTJ is always better than Hash-Join, a point that was also proved in the analysis. Importantly, this property doesn't hold for Yannakakis, which may perform redundant pre-processing in many cases.

S4 The paper is well-written and easy to follow.

== Weak Points for Improvement

W1. As the authors point out, there have been attempts to implement an approximate version of Yannakakis that, in contrast to the proposed algorithm, filter the relations up-front, yet in a fast, approximate way. Reference [24] is an example of that, but there are also others:

- Predicate transfer:
[Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. Y Yang, H Zhao, X Yu, P Koutris. CIDR 2024]

- Bitmap filters:
[Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server. Galindo-Legaria, C.A., Grabs, T., Gukal, S., Herbert, S., Surna, A., Wang, S., Yu, W., Zabback, P. and Zhang, S. ICDE 2008.]
Or a more modern reference:
[Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. K Lee, A Dutt, V Narasayya, S Chaudhuri. VLDB 2023]

This general strategy is not simply a matter of a deep system optimization as the authors seem to imply, but it represents an alternative approach, that, at least on the surface, is incomparable with the proposal of this paper. I believe that a comparison with a representative approach of that kind is necessary here, and the main item I would be looking for in a Revision.

W2. The implementation of tuple deletions should be discussed. What is the data structure used to store tuples that allows for fast deletion on-the-fly? This is a critical point when comparing the performance of TTJ with Hash-Join.

W3. The authors need to provide more details on the connection between their "tree convolution" and tree decompositions. The examples given all seem to be equivalent to generalized hypertree decompositions, and I think the only difference (in the definition) is that arbitrary levels of nesting are allowed. Why is that necessary and why do we need a new notion? Even if the authors can't prove a straightforward connection with existing width measures, they should provide examples that illustrate the difference and show the rationale behind their definition.

== Minor Issues:

- "Tree convolution" is perhaps an overloaded term. It also appears in learned query optimization:
[Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O. and Tatbul, N. Neo: a learned query optimizer. VLDB 2019.]

- The notation S[x] ion Figure 1 should be explained somewhere.

- Some more figures would be helpful, e.g., for the data in Example 1.1 or the join tree.

- In Definition 3.1, we also need every atom of the query to appear as a node.

- Typos:
  - line 258: Equation 2?
  - line 533: "consistent"
  - Caption of Figure 9c: "Another"

Additional Questions:
Relevance to Databases: Very High

Significance of Contribution: High

Readability and Organization: High

Fusion of Theory and Practice: Adequate

Length (Relative to the useful contents of the paper): Just Right

Please help ACM create a more efficient time-to-publication process: Using your best judgment, what amount of copy editing do you think this paper needs?: Light

Most ACM journal papers are researcher-oriented. Is this paper of potential interest to developers and engineers?: Yes