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

Georg Gottlob\textsuperscript{1}, Matthias Lanzinger\textsuperscript{1}, Davide Longo\textsuperscript{2}, Cem Okulmus\textsuperscript{3}, Reinhard Pichler\textsuperscript{2}, Alexander Selzer\textsuperscript{2}

\textsuperscript{1}University of Oxford
\textsuperscript{2}TU Wien
\textsuperscript{3}Umeå University
• Large join queries are becoming increasingly common
• The vast majority of user-generated queries are acyclic (ACQs) or almost-acyclic
• Intermediate results blow up with an increasing number of joins
• This can make even the otherwise easy-to-answer classes of queries challenging
• Yannakakis’ algorithm allows us to answer ACQs without unnecessary intermediate results
• No integration into standard relational database technologies has yet been performed
Acyclic Conjunctive Queries

- Conjunctive Queries (CQs) correspond to Relational Algebra-expressions of the form $\pi_U(R_1 \bowtie \ldots \bowtie R_n)$\(^1\), or SQL `SELECT-FROM-WHERE` queries.
- An Acyclic Conjunctive Query (ACQ) \(^2\) is a CQ that has a join tree.
- A join tree is a rooted, labelled tree $\langle T, r, \lambda \rangle$ with root $r$, such that:
  - $\lambda$ is a bijection that assigns to each node of $T$ one of the relations in $\{R_1, \ldots, R_n\}$ and
  - $\lambda$ 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 $u_j$.

---

\(^1\)where $R_1, \ldots, R_n$ are pairwise distinct and $U$ consists of all attributes in the $R_i$'s.

\(^2\)wrt. $\alpha$-acyclicity
Yannakakis’ algorithm

• It was shown that ACQs can be evaluated in time $O((||D|| + ||Q(D)||) \cdot ||Q||)$ using Yannakakis’ algorithm.

• Yannakakis’ algorithm for enumeration involves 3 traversals of the join tree $T$ which consist of
  1. a bottom-up traversal of semi-joins
  2. a top-down traversal of semi-joins
  3. a bottom-up traversal of full joins.
Query Evaluation in Practice

- Mostly based on the Volcano iterator model and System R architecture
- Execution as a sequence of two-way joins
- Optimization relies strongly on statistics and rules
- The tree-structure of ACQs is not fully taken into account
Query Evaluation in Practice

- PostgreSQL
  - \(< 12 \text{ FROM tables}\): exhaustive enumeration of query plans
  - → The global optimum wrt. statistics is very effective in most situations
  - \(\geq 12 \text{ FROM tables}\): search space explosion makes enumeration infeasible
  - → A heuristic approach based on a genetic algorithm is applied

- SparkSQL
  - Rule-based optimization
  - Currently supports only rudimentary cost-based optimization
  - Execution is highly parallelized

- DuckDB
  - In-memory query processing
  - Vectorized execution
  - Adaptive optimization for large queries
Yannakakis Query-Rewriting on top of the DBMS

- We introduce an approach to express the execution of Yannakakis’ algorithm via standard SQL statements (YANRE).
- Arbitrary join queries are translated into semi-joins and a sequence of full joins.
- This approach is applicable to any DBMS supporting temporary tables.
- The query engine does not have to be modified.
- Parallelization is possible.
SELECT release.id, release_group.type, artist_credit.id,
release.release_group, release_status.id
FROM release_group, artist_credit, release, track, release_status,
release_group_secondary_type_join, release_unknown_country,
release_group_primary_type
WHERE release_group.artist_credit = artist_credit.id
AND release_group.id = release.release_group
AND artist_credit.id = track.artist_credit
AND release.status = release_status.id
AND release_group.id = release_group_secondary_type_join.release_group
AND release.id = release_unknown_country.release
AND release_group.type = release_group_primary_type.id;
Yannakakis Query-Rewriting: Example

DuckDB

\[ \pi \]

0.2s

# 158593222

\[ \bowtie \]

273.5s

# 158593222

\[ \bowtie \]

307.61s

# 2794874903

\[ \bowtie \]

0s

# 52536

release_group

0s

# 2575238

release_group_prim.

0s

# 5

track

0.4s

# 37738957

artist_credit

0s

# 2328626

release_unknown

0s

# 228699

release_status_sec.

0s

# 639242

release

0s

# 3283476

release_status

0s

# 6

release_status_sec.

0s

# 748101

track

0.4s

# 37738957

artist_credit

0s

# 2328626

release_unknown

0s

# 228699

release_status_sec.

0s

# 748101

release

0s

# 3283476

release_status

0s

# 6

release_status_sec.

0s

# 748101

release

0s

# 3283476

release_status

0s

# 6

release_status_sec.

0s

# 748101

release

0s

# 3283476
Yannakakis Query-Rewriting: Example

Yannakakis Join Stage

\[ \pi \ 0.6s \]
\[ \# \ 158593222 \]
\[ \bowtie \ 15.5s \]
\[ \# \ 158593222 \]
\[ \bowtie \ 15.1s \]
\[ \# \ 149184804 \]
\[ \bowtie \ 0s \]
\[ \# \ 45112 \]
\[ \bowtie \ 0s \]
\[ \# \ 45256 \]
\[ \bowtie \ 0s \]
\[ \# \ 45256 \]
\[ \bowtie \ 0s \]
\[ \# \ 3132087 \]
\[ \bowtie \ 0s \]
\[ \# \ 201214 \]
\[ \bowtie \ 0s \]
\[ \# \ 45256 \]
\[ \bowtie \ 0s \]
\[ \# \ 6 \]
\[ \bowtie \ 0s \]
\[ \# \ 43806 \]
\[ \bowtie \ 0s \]
\[ \# \ 18843 \]
\[ \bowtie \ 0s \]
\[ \# \ 45112 \]
\[ \bowtie \ 0s \]
\[ \# \ 45112 \]
\[ \bowtie \ 0s \]
\[ \# \ 10489038 \]
\[ \bowtie \ 0s \]
\[ \# \ 18843 \]
\[ \bowtie \ 0s \]
\[ \# \ 43806 \]
\[ \bowtie \ 0s \]
\[ \# \ 45112 \]
\[ \bowtie \ 0s \]
\[ \# \ 10489038 \]
\[ \bowtie \ 0s \]
\[ \# \ 18843 \]
\[ \bowtie \ 0s \]
\[ \# \ 43806 \]
\[ \bowtie \ 0s \]
\[ \# \ 45112 \]
\[ \bowtie \ 0s \]
\[ \# \ 10489038 \]
<table>
<thead>
<tr>
<th>System / Stage</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DuckDB</td>
<td>151.9 s</td>
</tr>
<tr>
<td>DuckDB + YANRE / Total</td>
<td>42.1 s</td>
</tr>
<tr>
<td>DuckDB + YANRE / Setup</td>
<td>0.5 s</td>
</tr>
<tr>
<td>DuckDB + YANRE / ⋈-up</td>
<td>1.5 s</td>
</tr>
<tr>
<td>DuckDB + YANRE / ⋈-down</td>
<td>0.9 s</td>
</tr>
<tr>
<td>DuckDB + YANRE / Join</td>
<td>39.2 s</td>
</tr>
</tbody>
</table>
Performance Evaluation

- Random-walk queries by Mancini et al.\(^3\)
  - Based on the musicbrainz dataset
  - 435 challenging synthetic queries
  - 351 acyclic queries
  - joins between 2 and 30 tables

- 3 DBMSs
  - PostgreSQL
  - SparkSQL
  - DuckDB

\(^3\)Efficient massively parallel join optimization for large queries, SIGMOD 2022
Performance Evaluation

• 351 full enumeration queries

```
SELECT area_type.id, iso_3166_3.area FROM area_type, area, iso_3166_2, iso_3166_3
WHERE area_type.id = area.type AND
area.id = iso_3166_2.area AND
area.id = iso_3166_3.area;
```

• 351 min-aggregate queries (OMA)

```
SELECT min(area.type)
FROM area_type, area, iso_3166_2, iso_3166_3
WHERE area_type.id = area.type AND
area.id = iso_3166_2.area
AND area.id = iso_3166_3.area;
```
Table 1: DuckDB, PostgreSQL, and Spark SQL with or without YANRE over acyclic queries on the MusicBrainz dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Timeouts</th>
<th>Max</th>
<th>Mean</th>
<th>Med.</th>
<th>Std.Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DuckDB</td>
<td>69</td>
<td>770.55</td>
<td>252.27</td>
<td>0.67</td>
<td>473.87</td>
</tr>
<tr>
<td>DuckDB+YANRE</td>
<td>29</td>
<td>801.79</td>
<td>121.39</td>
<td>2.34</td>
<td>335.75</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>97</td>
<td>1107.66</td>
<td>364.32</td>
<td>4.02</td>
<td>533.47</td>
</tr>
<tr>
<td>PostgreSQL+YANRE</td>
<td>70</td>
<td>786.31</td>
<td>283.2</td>
<td>25.71</td>
<td>470.2</td>
</tr>
<tr>
<td>SparkSQL</td>
<td>87</td>
<td>1164.06</td>
<td>358.28</td>
<td>23.91</td>
<td>513.67</td>
</tr>
<tr>
<td>SparkSQL+YANRE</td>
<td>29</td>
<td>876.74</td>
<td>204.11</td>
<td>59.45</td>
<td>335.47</td>
</tr>
</tbody>
</table>

1. Excludes timeout values.
2. Timeout treated as 1200 seconds.
Table 2: DuckDB, PostgreSQL, and Spark SQL with or without YANRE over acyclic queries on the MusicBrainz dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Timeouts</th>
<th>Max</th>
<th>Mean</th>
<th>Med.</th>
<th>Std.Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DuckDB</td>
<td>58</td>
<td>1169.38</td>
<td>217.9</td>
<td>0.44</td>
<td>447.94</td>
</tr>
<tr>
<td>DuckDB+YANRE</td>
<td>0</td>
<td>15.57</td>
<td>2.31</td>
<td>1.44</td>
<td>2.38</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>91</td>
<td>1131.08</td>
<td>342.78</td>
<td>2.82</td>
<td>524.16</td>
</tr>
<tr>
<td>PostgreSQL+YANRE</td>
<td>2</td>
<td>236.75</td>
<td>24.74</td>
<td>5.83</td>
<td>93.73</td>
</tr>
<tr>
<td>SparkSQL</td>
<td>91</td>
<td>1082.58</td>
<td>365.76</td>
<td>25.35</td>
<td>518.7</td>
</tr>
<tr>
<td>SparkSQL+YANRE</td>
<td>3</td>
<td>214.04</td>
<td>41.12</td>
<td>16.14</td>
<td>113.24</td>
</tr>
</tbody>
</table>
• We have seen that a structure-guided query execution approach can radically reduce the amount of timeouts in large join queries.

• A lightweight rewriting on top of the DBMS was shown to be feasible despite the overhead.

• Even though standard query optimizers are hard to beat in most cases, on large queries they can fail spectacularly.

• We have identified a class of queries (OMA) which is especially well-suited for structure-guided query-optimization.
Open Problems

- Benchmarks for large join queries
- Deep integration into the query optimizer
- Combining structural and quantitative query optimization
- Extending the class $OMA$
Questions?