LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins

Bernhard Radke, Thomas Neumann

Technische Universität München
Join Ordering

SELECT ...
FROM A, B, C, D, E, F
WHERE A.a = B.a AND B.b = C.b AND B.c = E.c
    AND C.d = D.d AND E.e = F.e
Join Ordering is NP-Hard

Tableau (DBTEST 2018): Queries regularly involve a few dozen joins
SAP (BTW 2017): Largest query touches 4,598 relations
Adaptive Optimization of Very Large Join Queries (SIGMOD 2018)

For performance and correctness reasons: Do not consider crossproducts
Search Space Linearization

- If the order of relations in the optimal plan is known
- Generating the optimal plan from this linearization takes polynomial time
- Optimally combine optimal solutions for subchains
Search Space Linearization

- Of course: Optimal order unknown
- But IKKBZ (TODS 3/1984, VLDB 1986): optimal left-deep plan in $O(n^2)$
- Using IKKBZ to linearize the search space yields good bushy plans
Requires acyclic query graph (build MST if cyclic)

Idea: Transform precedence graphs into a linear order

Assign ranks to nodes (cost/benefit ratio)

Successively merge child chains increasing in ranks

Resolve contradictory sequences in child chains by merging them into a single node
Build precedence graph (here rooted in A)

Resolve contradictory sequences in child chains by merging them into a single node
rank(E) > rank(F), but E has to precede F

Merge child chains increasing in the nodes rank
rank(C) < rank(E,F) < rank(D)
Search Space Linearization

Query Graph

A → B
  C → D
  E → F

Linearized Search Space

A → B → C
  E → F → D

- Repeat this for each relation
- Guarantee: Final plan at least as good as the best left-deep plan
Solve easy cases **optimally**

Search Space Linearization: near-optimal plans for common queries

Gracefully tune down plan quality for the most complex queries

Optimize queries on hundreds of relations in the **blink of an eye**
Adaptive Optimization of Very Large Join Queries (SIGMOD 2018)

Query

- easy?
  - yes: solve optimally (DPhyp)
  - no
    - corner case?
      - yes: cannot linearize (GOO/DPhyp)
      - no
        - medium?
          - yes: search space linearization (LinDP++)
          - no: gracefully introduce greediness to keep optimization time reasonable (GOO/linDP)

Non-Inner Joins – More Than a Corner Case

- Tableau (DBTEST 2018): 20% of the queries involve outer joins, up to 247 in a single query
- Others also report significant numbers of queries with outer joins
- Non-Inner joins impose reordering constraints
- Expressed using hyperedges (Moerkotte et al. SIGMOD 2013)
IKKBZ only handles regular graphs
Still: Given a proper linearization, polynomial time construction of bushy plan
How to extend IKKBZ to generate linearizations for hypergraphs?
Precedence for Hypergraphs

- Hyperedge \{C,D\} – \{E\}
- Backward and forward hyperedges
Precedence for Hypergraphs – Backward Hyperedges

- Precedence DAG, multiple relations have to precede
- During merge: Ensure all precedence constraints are satisfied
Join towards multiple relations, no left deep solution
- Recursively linearize group \{C,D\}: C,B,D
- Guarantee: Final plan at least as good as the best left-deep plan if there exists one
Experiments

- More than 10 different join ordering algorithms
- 60 seconds timeout per query
- Standard benchmarks (TPC-H, TPC-DS, etc.) easily optimized by full DP

⇒ 1,000 realistic random tree queries
  - Up to 100 relations each
  - Random reordering constraints
Plan Quality

- Cost normalized to the best known plan per query

- LinDP++ generates clearly superior plans
Optimization Time

- Pure inner join queries vs. queries with outer joins

LinDP++ handles non-inner joins as fast as inner joins
Adaptive Optimization of Very Large Join Queries (SIGMOD 2018)

For performance and correctness reasons: Do not consider crossproducts
Do Not Consider Crossproducts

1. Performance
   - Exponential search space regardless of the query’s structure
   - Most considered crossproducts will not reduce cost \((A \times B \in \mathcal{O}(|A||B|))\)

2. Correctness
   - Crossproducts in the presence of non-inner joins can yield wrong query results
Do Not Consider Crossproducts

1. Performance
   - Exponential search space regardless of the query’s structure
   - Most considered crossproducts will **not reduce cost** \( (A \times B \in \mathcal{O}(|A||B|)) \)

2. Correctness
   - Crossproducts in the presence of non-inner joins can yield wrong query results
Do Consider Some Crossproducts

- Observation: Some plans would significantly benefit from crossproducts
- TPC-DS: Crossproducts improve geometric mean of cost by 15%
- However: 82% of the queries do not benefit at all from crossproducts
- Thus: Do consider some crossproducts (ideally the important ones)

- How to efficiently discover the valid and important crossproducts?
Intuitively: Crossproduct to avoid massive intermediate results
That is: Bypass expensive joins
Idea: Check neighboring inner joins for opportunities

If crossproduct is smaller than both intermediate results:
Add explicit edge to the query graph
Cost Improvement

Cost Improvement Factor

Crossproducts
- None
- All
- Heuristic
## Optimization Overhead

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>TPC-H</th>
<th>TPC-DS</th>
<th>LDBC</th>
<th>JOB</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinDP++</td>
<td>8%</td>
<td>6%</td>
<td>0</td>
<td>8%</td>
<td>0</td>
</tr>
<tr>
<td>DPhyp</td>
<td>12%</td>
<td>2.8X</td>
<td>0</td>
<td>76%</td>
<td>0</td>
</tr>
<tr>
<td>All Crossproducts</td>
<td>2.4X</td>
<td>214X</td>
<td>53X</td>
<td>83X</td>
<td>152X</td>
</tr>
</tbody>
</table>

- LinDP++ efficiently considers most of the relevant crossproducts
LinDP++

Optimize as fast as pure inner join queries

Generate significantly better plans

Efficiently consider promising crossproducts
Bonus Slides
Standard Benchmarks

- **Plan Quality (normalized cost)**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>TPC-H</th>
<th>TPC-DS</th>
<th>LDBC</th>
<th>JOB</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPhyp</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>LinDP++</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.07</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- **Optimization Time (ms)**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>TPC-H</th>
<th>TPC-DS</th>
<th>LDBC</th>
<th>JOB</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPhyp</td>
<td>0.4</td>
<td>90</td>
<td>1.2</td>
<td>227</td>
<td>2.2K</td>
</tr>
<tr>
<td>linearized DP</td>
<td>1.4</td>
<td>18.7</td>
<td>4.4</td>
<td>33.4</td>
<td>4.7K</td>
</tr>
<tr>
<td>LinDP++</td>
<td>1.6</td>
<td>19.9</td>
<td>4.4</td>
<td>36.2</td>
<td>4.7K</td>
</tr>
</tbody>
</table>

- Standard benchmarks *barely a challenge* for an optimizer