Multi-Query Optimization

Prasan Roy
S. Seshadri
S. Sudarshan
Siddhesh Bhobe
Aditya Phatak

Computer Science and Engineering Department
I.I.T. Bombay

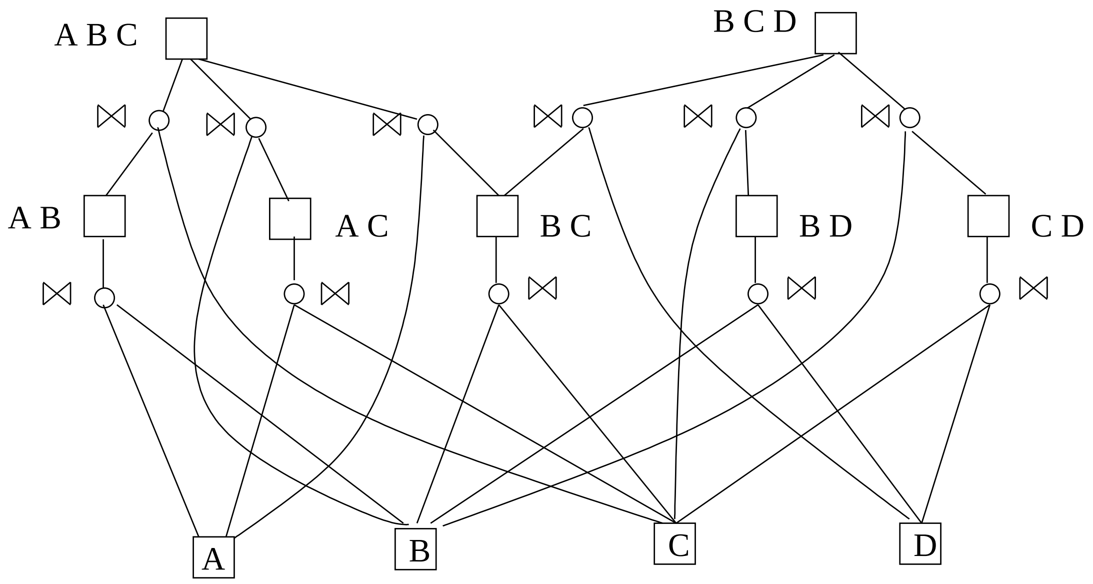
Why Multi-Query Optimization

- To tackle complex decision support queries
  - Sequence (batch) of related queries
  - Queries with common subexpressions
    - * Esp. with use of views and WITH clause
    - * Multi-output operators
      E.g. Engage.Fusion (now Informix Formation)
- Data warehouses, OLAP, ...

Multi-Query Optimization

- Given: query, or a batch of queries
- Goal: exploit common subexpressions
- Problem: many alternative query plans with different costs
- E.g.:
  - $Q_1 : (R \bowtie S) \bowtie P$ and $Q_2 : (R \bowtie T) \bowtie S$ – no common subexpressions
  - Rewrite Q_2: $(R \bowtie S) \bowtie T$ – now $\exists$ common subexpression
  - Can compute $R \bowtie S$, store (materialize) and reuse
- Problem: expression form with CSE may or may not be cheaper!
- Problem: lots of alternative plans $\Rightarrow$ expensive

AND-OR DAG Representation



- □: *equivalence* nodes
- ○: *operation* nodes

Steps in Multi-Query Optimization

- Recognize possibilities of sharing
  - Extend Volcano's Query-DAG generation algorithm
- Find best plan taking sharing into account
  - Local best plans may not be globally optimal
  - Exhaustive algorithms are doubly exponential
  - Need good heuristics

Developed Heuristics

- Volcano-SH
  - queries optimized independently using Volcano
  - sharing determined among the best plans
- Volcano-RU
  - queries optimized using Volcano considering previous queries in the sequence
- Greedy
  - iterative, at each step pick the “best” intermediate result to materialize

The Greedy Heuristic

1. Compute DAG and best plans for each node using Volcano
2. Find *benefit* of materializing each unmaterialized node
   - benefit of n = reduction in total cost due to materializing n
3. Choose node with max cost benefit
   - mark it, materialize it and set its cost to reuse-cost
   - recompute best plans
4. Repeat from Step 2, until no decrease in cost

Incremental Cost Update Propagation

- Recomputing best plans from scratch on each benefit computation very expensive
- Observation: *Marking a node as materialized only affects its ancestors*
- Incremental Propagation:
  1. Start from marked node, and propagate cost changes up the DAG
  2. If cost of a node does not change, no need to propagate changes further up
- Use topological sort numbering to decide which node to propagate changes to next (*ensures no revisits*)

Improvements to Greedy: Monotonicity Heuristic

- **Monotonicity Assumption:** Benefit of materializing a node never increases as other nodes are materialized.
- Monotonicity $\Rightarrow$ *old benefit estimates are overestimates on current benefit*
- "old" $\equiv$ computed in an earlier iteration of greedy
- *Key idea*: Keep heap of nodes sorted on benefit *overestimates*
  - Repeat forever
    1. Pop top node t
    2. If benefit t is ≤ 0, break
    3. if benefit of t is *current*, t is the node with maximum benefit; delete from heap and materialize it
    4. else update its benefit, mark as *current*, reinsert in heap
- Monotonicity generally holds; reduces opt. run time by 90%

Performance Study

- Implemented all algorithms
  (17K lines Volcano + 3K lines MQO)
- Benchmarks
  - Stand-Alone TPCD (database size 1GB)
  - Batched TPCD
  - Scaleup with number of queries
  - Estimated cost + MS SQL-Server 6.5 runtimes

Performance Results

- Greedy is best overall, followed by Volcano-RU and Volcano-SH
  - Improvements over Volcano for Batched TPCD:
    Volcano-SH, Volcano-RU: upto 14%; Greedy: upto 56%
- Volcano-SH, Volcano-RU have negligible overheads.
  Overhead for Greedy acceptable for complex expensive queries
- Benefits show up on TPCD running on MS SQL-Server 6.5
  - 50% improvement in execution time for Query 11
  - significant absolute improvements for Query 2, Query 15
- Greedy shows linear scaleup with number of queries

Extensions

- Incorporating index selection
  - Index considered as a physical property (like sort order)
- Handling nested queries
  - Extend DAG to represent nested queries
  - Correlated results are not considered for materialization
- Default Materialization
  - Some results may be available as materialized on computation (e.g. sort operator result)

Future Work

- Large workloads: Query Abstraction and Parameterized Queries
  - Similar queries can be encoded as parameterized queries
- Restricting Search Space to Linear Plans
  - Motivation: Reduces the search space
  - Initial search space has only linear trees
  - On subsequent materializations, the search space is expanded
- Space Constraints
  - Consider benefit per unit space instead.
- Scheduling
  - Try to avoid materialization by sharing scans