Projection Pushing Revisited

EDBT 2004

Ben McMahan, Guoqiang Pan, Patrick Porter, Moshe Vardi
Rice University

March 16, 2004

Overview

- **Review and Motivation**
- Experimental Setup
- Structural Optimizations
- Experimental Results
- Conclusions

What is Query Optimization?

- Queries are written to access the data in a database.
- Queries can be transformed to logically equivalent queries
- Not all equivalent queries are equal:
  - $(r1 \bowtie r2) \bowtie \emptyset$, vs. $(\emptyset \bowtie r1) \bowtie r2$
  - $\pi_a(r1 \bowtie_a r2)$, vs. $(\pi_a r1) \bowtie (\pi_a r2)$
- We call a particular method of execution a **plan**
- Databases typically use cost-based optimization

What is Cost-Based Optimization?

Cost-based optimization is a search technique that requires

- A search space of plans,
- A **cost estimation** method for each plan, and
- An **enumeration** algorithm.

Typically, information about the database is used to assign a cost to each operation.

Goal is to find an accurate cost estimation method and an efficient enumeration algorithm to find a low cost plan

Problems with Cost-Based Optimization

- Problems arise when the number of joins is large
- For n joins, there are $O(n!)$ possible plans
- Dynamic programming and the principle of optimality reduce this to $O(n2^{n-1})$
- Thus, cost-based optimization does not scale.

Where Might This Be a Problem?

Queries with a large number of joins start appearing in

- Mediation systems,
- Complex views joined with other complex views, and
- Machine generated queries.

All of these domains are continually growing in use.

An Alternative Approach: Structural Heuristics

Structural Heuristics

- Focus on optimizing structural properties of the query
- Minimize the arity of the intermediate tables
- Constant arity bound $\rightarrow$ polynomial size bound
- Minimal arity is directly related to the treewidth of the join graph

- Review and Motivation
- **Experimental Setup**
- Structural Optimizations
- Experimental Results
- Conclusions

Experiment Setup

We challenged the effectiveness of cost-based optimization with

- Small databases – One table with two attributes and six tuples
- Large queries – Hundreds of joins
- Focused on Project-Join queries.
- Consider Boolean queries (output is empty or non-empty)

To achieve all this we generated queries from 3-COLOR problems.

3-COLOR

An instance of 3-COLOR is a

- Graph $G = (V, E)$, $|V| = n$ and $|E| = m$, and a
- Set of colors $C = \{1, 2, 3\}$.

The problem is whether or not there is a way to color V using C where for every $(u, v) \in E$, $c(u) \neq c(v)$.

3-COLOR as a Query

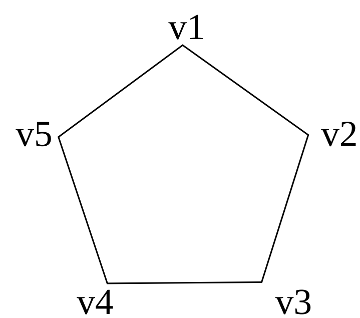
We define an **EDGE** relation containing all pairs of distinct colors:

EDGE	
1	2
1	3
2	1
2	3
3	1
3	2

EDGE contains all 3-colorable colorings of an edge. Our query is then

$$Q_G = \pi_{\emptyset} \bowtie_{(u,v) \in E} EDGE(u, v)$$

Pentagon Example



A pentagon is a graph $G = (V, E)$ where $V = \{v1, v2, v3, v4, v5\}$ and $E = \{(v1, v2), (v1, v5), (v2, v3), (v3, v4), (v4, v5)\}$

So the corresponding query would be:

$$Q_G = \pi_{\emptyset} EDGE(v1, v2) \bowtie EDGE(v1, v5) \bowtie EDGE(v2, v3) \bowtie EDGE(v3, v4) \bowtie EDGE(v4, v5)$$

Our Approach

Using PostgreSQL 7.1.3, for each graph,

- We construct an SQL query
- Run the query
- Gather results and both optimization and execution time

- Review and Motivation
- Experimental Setup
- **Structural Optimizations**
- Experimental Results
- Conclusions

Naive Query

The **naive** query is the most direct translation to SQL.

The pentagon example would yield:

```
SELECT 1
WHERE EXISTS (
SELECT *
FROM EDGE e1 (v1,v2), EDGE e2 (v1,v5), EDGE e3 (v2,v3), EDGE e4
(v3,v4), EDGE e5 (v4,v5)
WHERE e1.v1 = e2.v1
AND e1.v2 = e3.v2
AND e2.v5 = e5.v5
AND e3.v3 = e4.v3
AND e4.v4 = e5.v4);
```

Straightforward Query

The **straightforward** query explicitly lists the join order.

The pentagon example is now:

```
SELECT 1
WHERE EXISTS (
SELECT *
FROM EDGE e5 (v4,v5) NATURAL JOIN (
    EDGE e4 (v3,v4) NATURAL JOIN (
        EDGE e3 (v2,v3) NATURAL JOIN (
        EDGE e2 (v1,v5) NATURAL JOIN EDGE e1 (v1,v2)))));
```

Naive vs Straightforward

- NATURAL JOIN assumes equality on same names
- Execution time the same as Naive
- Compilation time decreased by 3 orders of magnitude
- Neither naive nor straightforward plans use early projection!
  - This is also true of DB2 and Oracle

Early Projection

Our queries have the form $\pi_{v_1,\ldots,v_k}(r_1 \bowtie \ldots \bowtie r_m)$.

If a vertex $v_j \notin \{r_{q+1}, \ldots, r_m\}$, then we can rewrite the query into:

$$\pi_{v_1,\ldots,v_k}(\pi_{livevars}(r_1 \bowtie \ldots \bowtie r_q) \bowtie r_{q+1} \bowtie \ldots \bowtie r_m)$$

- livevars contains all the variables except v_j
- v_j has been **projected early**
- Arity of intermediate results has been reduced

Early Projection Continued

Our pentagon example now looks like:

```
SELECT 1
WHERE EXISTS (
SELECT *
FROM edge e5 (v4,v5) NATURAL JOIN (
    SELECT e4.v4, t3.v5
    FROM edge e4 (v3,v4) NATURAL JOIN (
        SELECT e3.v3, t4.v5
        FROM edge e3 (v2,v3) NATURAL JOIN (
            SELECT e1.v2, e2.v5
            FROM edge e2 (v1,v5) NATURAL JOIN edge e1 (v1,v2)
            ) AS t4 ) AS t3 ) AS t2
);
```

Reordering Relations

Reordering relations can help us project early more aggressively. For example,

- Let v_1 be only in r_1 and r_m.
- Then v_1 will not be projected early
- But v_1 could be projected out after 1 join.

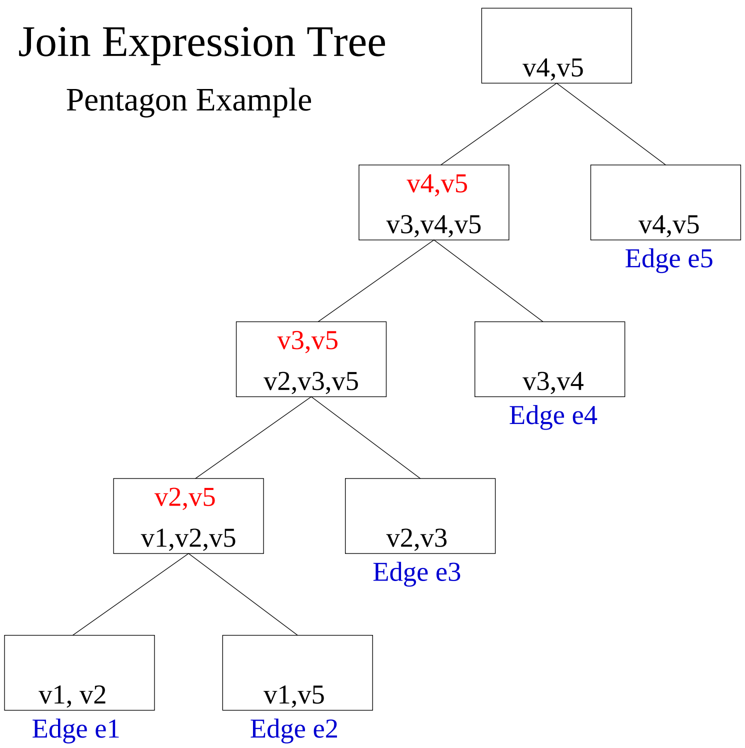
Greedy Heuristic

Finding an optimal relation order is hard so we permute the relations greedily

- Computing the order incrementally
- At each step, look for relation that would project early the most attributes
- To break ties, choose the relation that shares the least attributes with the remaining relations
- Further ties are broken randomly

Limits?

What are the limits of early projection?



Theoretical Results

Let **joinwidth** of a query Q be the smallest width of all possible join expression trees

Then, the joinwidth of the query is the treewidth of its join graph plus one.

The **join graph** of a query creates a vertex for every attribute and a clique between every relation.

Treewidth is a

- Notion that formalizes how tree-like a graph is
- Can be defined through **treedecompositions**

Central Theorem

Theorem 1: Given a project-join query Q, the joinwidth of Q is equal to the treewidth of the joingraph of Q plus one.

Proof Sketch:

Lemma 1: Given a project-join query Q and a join expression tree J_Q of width k, there is a tree decomposition $T_{J_Q} = ((I, F), X)$ of the join graph G_Q such that the width of T_{J_Q} is $k - 1$.

Lemma 2: Given a project-join query Q, and join graph G_Q, and a tree decomposition of G_Q of treewidth k, there is a join expression tree of Q with width $k + 1$

Bringing It All Together

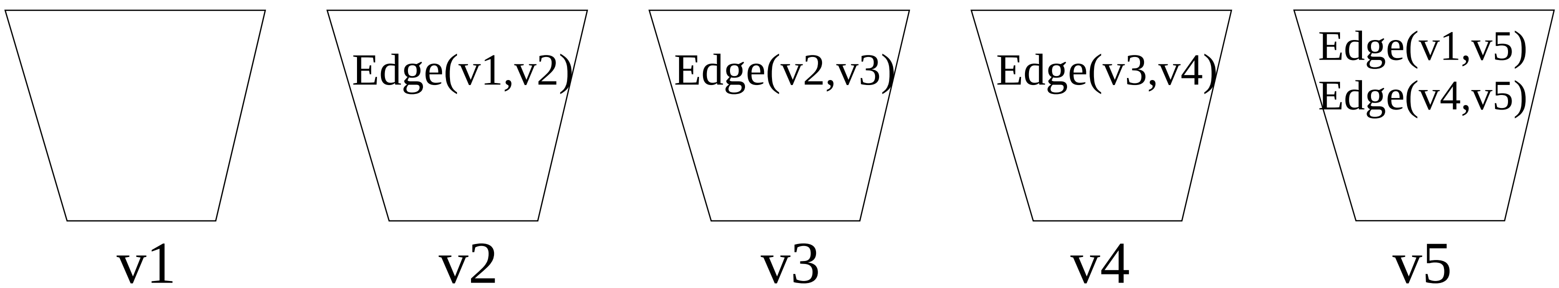
Algorithms for finding small treewidths should work for query optimization.

Artificial Intellegence uses a technique called **bucket elimination**

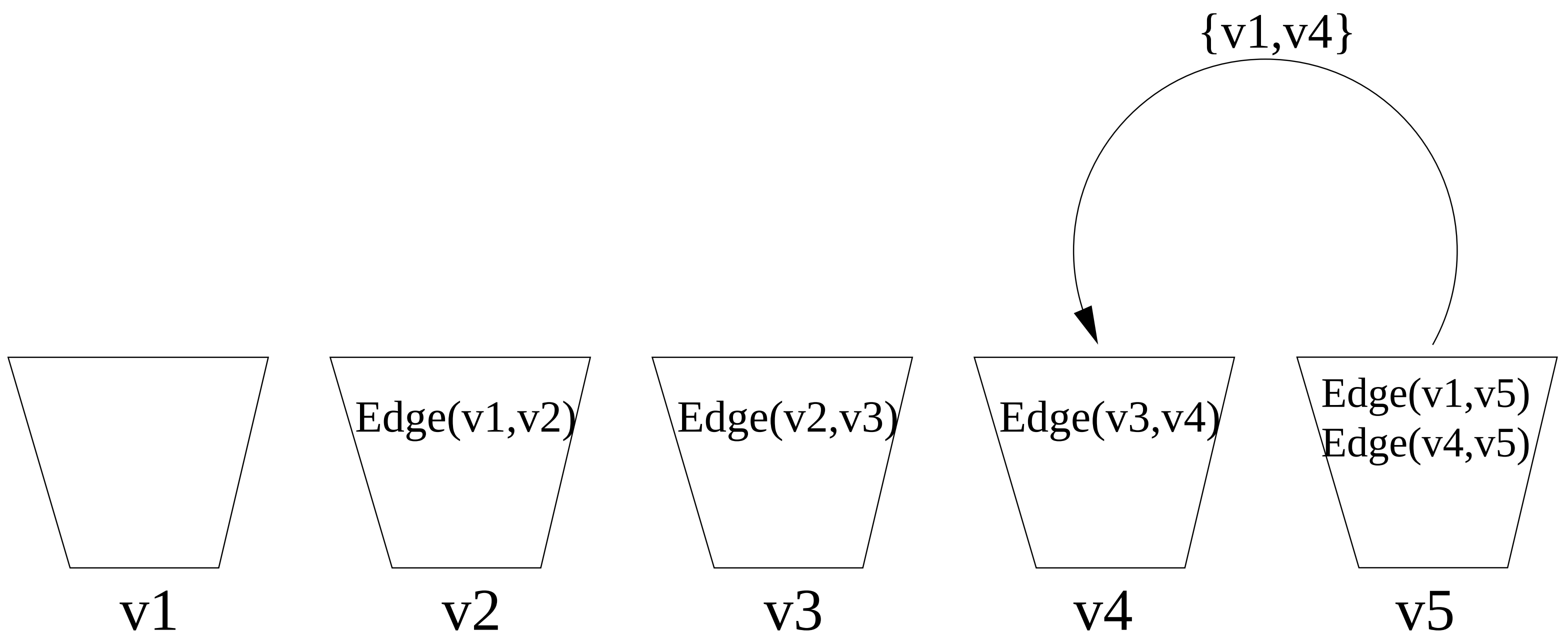
- A bucket is made for each attribute in the query
- Given an order of the attributes, relations are placed into the highest labeled bucket
- The bucket is processed and associated attribute projected out
- The results are then placed in the next highest bucket

Given an suitable order this method will obtain an optimal solution.

Bucket Elimination



Bucket Elimination after one step



Maximum Cardinality Search

We used the Maximum Cardinality Search (MCS) order to fuel the bucket elimination method

- Iterating over the join graph
- Each iteration picks the attribute most connected to those already chosen
- Ties broken arbitrarily

MCS has been used successfully in constraint satisfaction

Other attribute orders are explored later in the talk

- Review and Motivation
- Experimental Setup
- Structural Optimizations
- **Experimental Results**
- Conclusions

Random Queries

We generate random 3-COLOR graphs using two parameters

- The **order** – number of vertices
- The **density** – number of edges / vertices
- Two distinct vertices are picked uniformally
- Edges are created, without repetition, until all edges have been generated

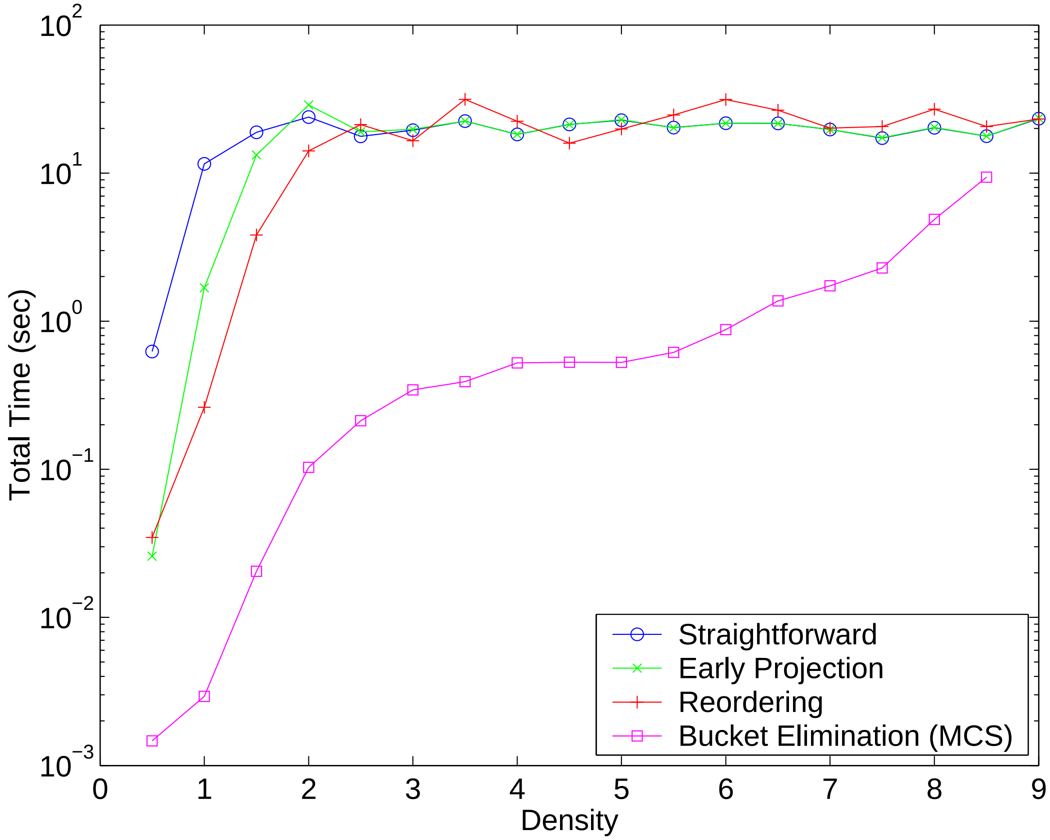
Scaling

We are concerned with two type of scalability

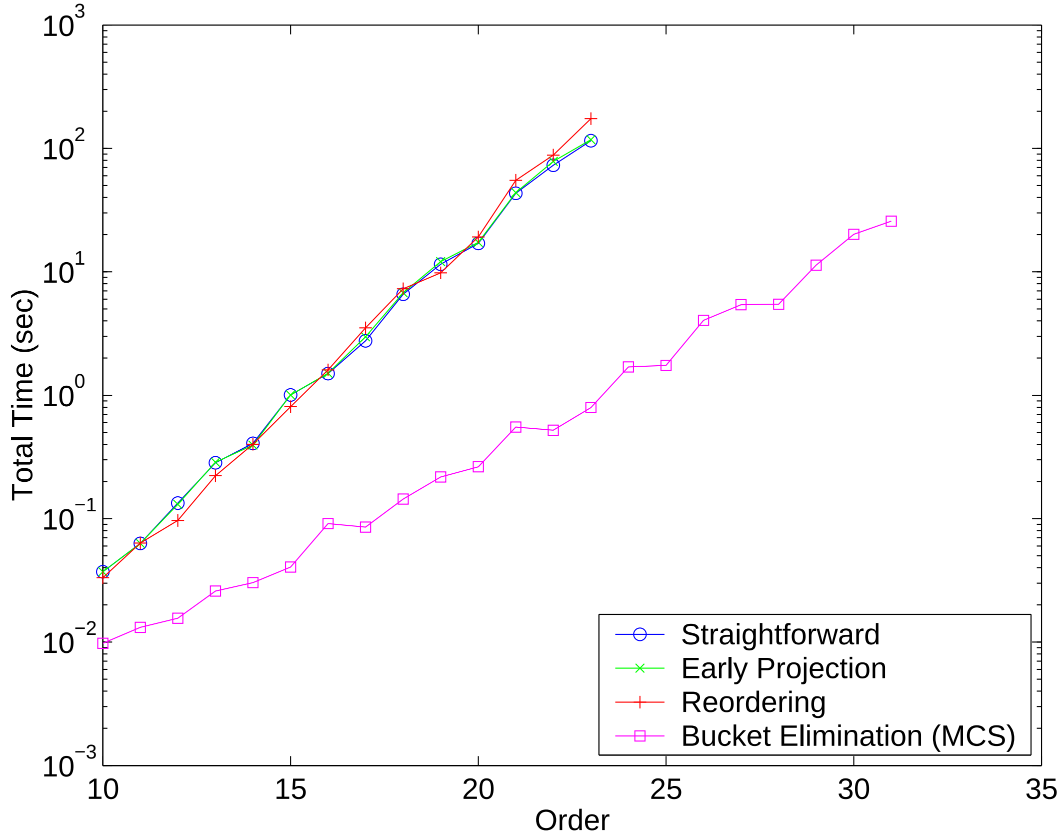
- Density scaling – Fix the order of the queries and increase the density
  - Tests scalability over structural changes in the query
  - Move from underconstrained to overconstrained instances
- Order scaling – Fix the density of the query and increase the order
  - Tests tradition scalability of optimization

For each order and density, 100 graphs are generated and the median execution time is plotted.

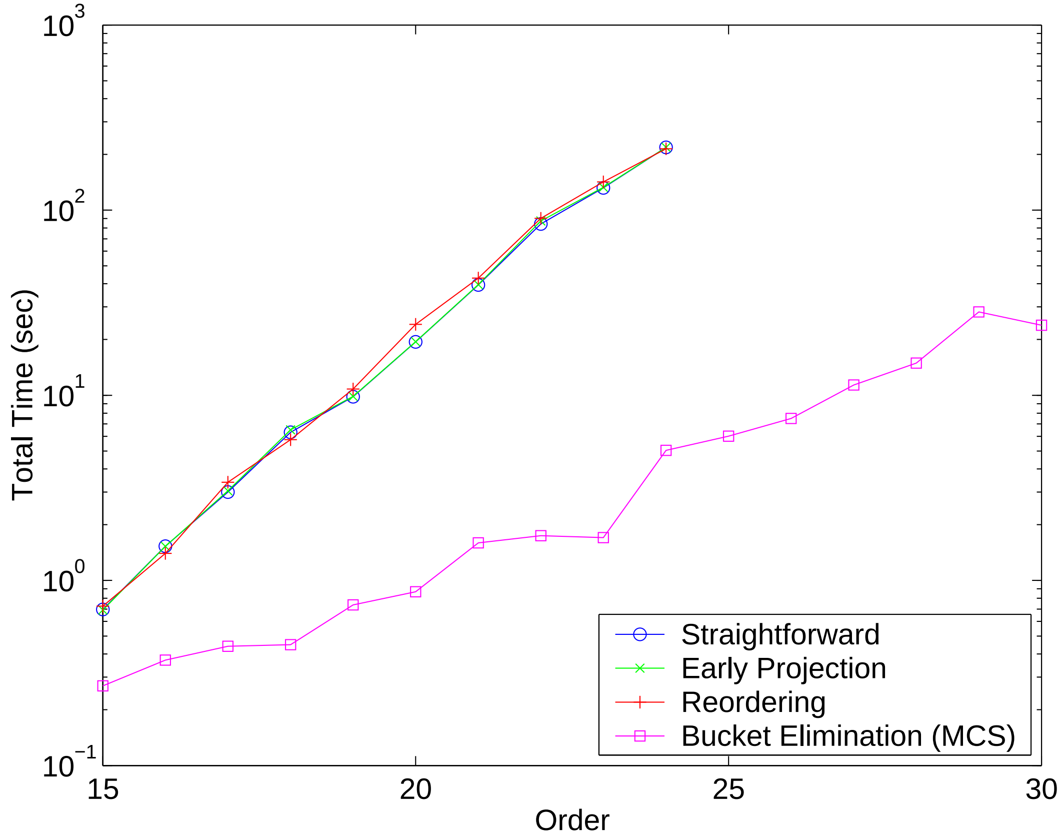
Density Scaling - Order 20 - Logscale



Order Scaling - Density 3.0 - Logscale

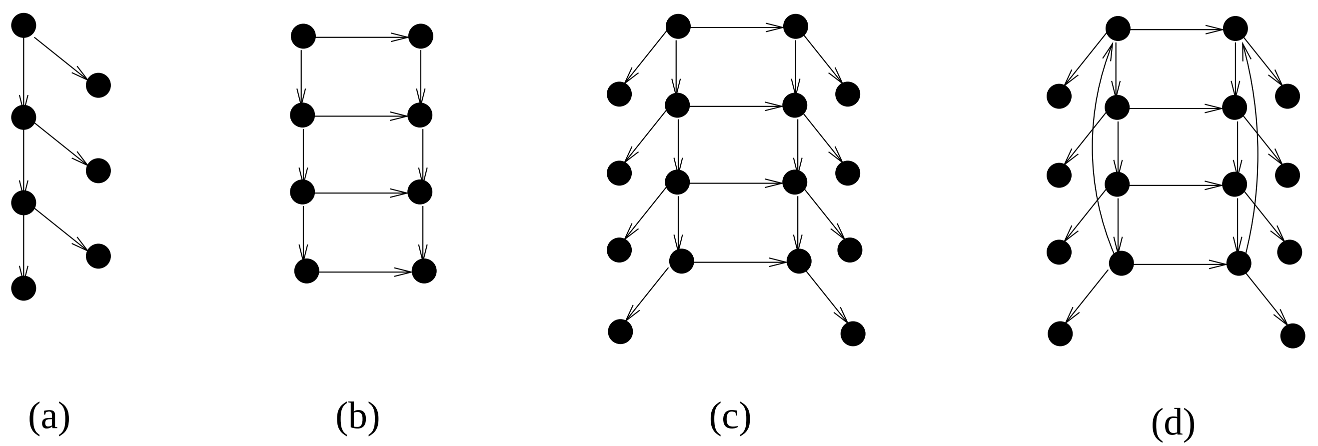


Order Scaling - Density 6.0 - Logscale



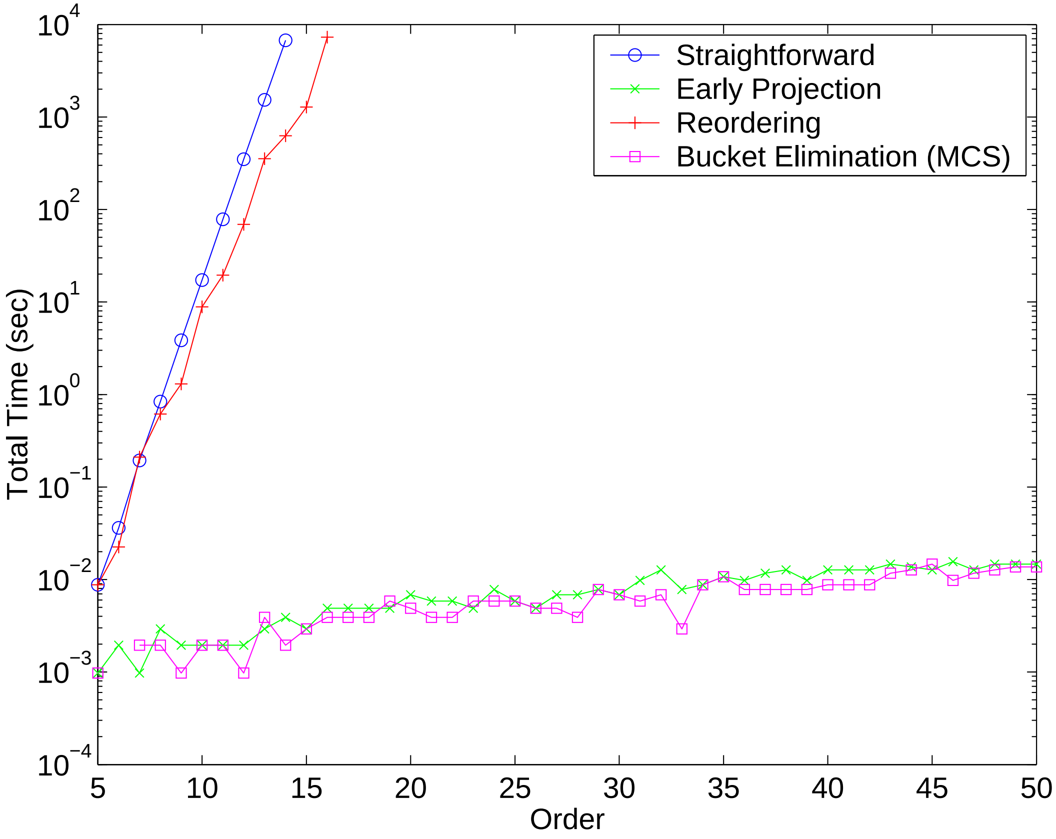
Structured Queries

We also used structured queries

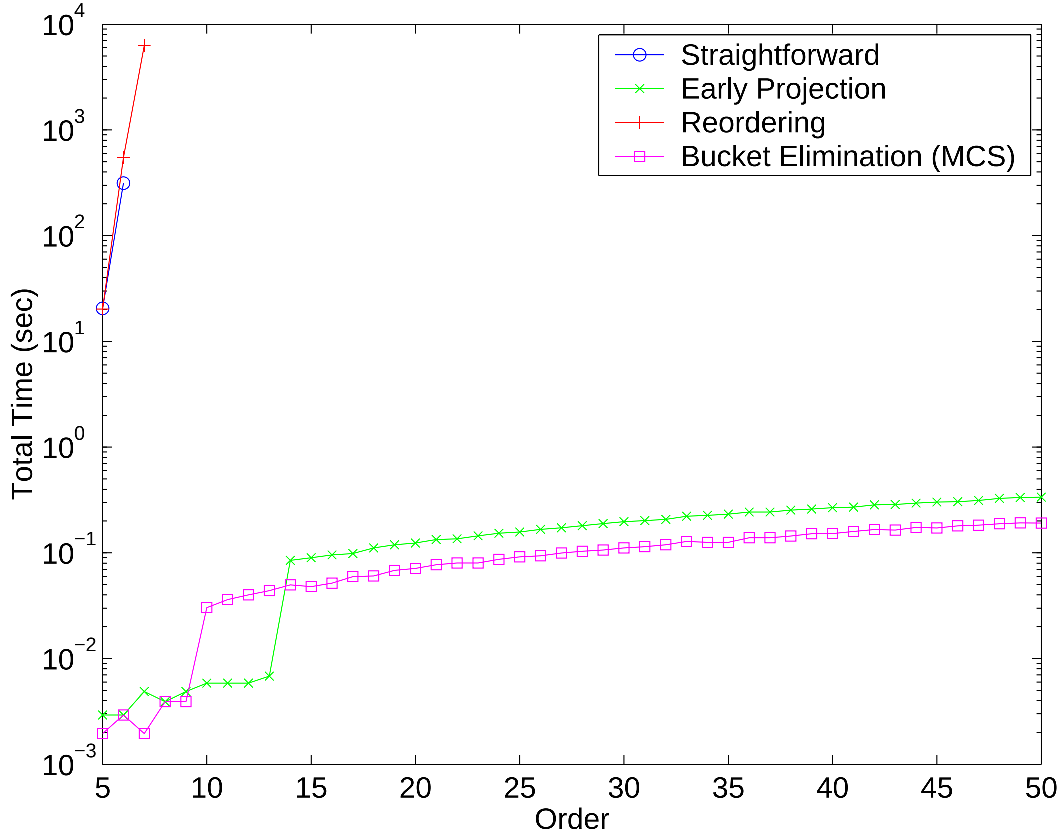


(a) Augmented Path (b) Ladder (c) Augmented Ladder (d) Augmented Circular Ladder

Augmented Path - Logscale



Augmented Circular Ladder - Logscale



- Review and Motivation
- Experimental Setup
- Structural Optimizations
- Experimental Results
- **Conclusions**

Conclusions

- Early projection, applied greedily, can provide exponential improvement over straightforward approaches

- Bucket elimination provides another exponential improvement.

- Structural heuristics can be used to optimize queries successfully

Note that our results also hold for non-Boolean queries and our methods work for more general queries, not just 3-COLOR.

Future Work

- Find a framework in which to combine cost-based and structural techniques, i.e. weighted graphs or width as a cost measurement
- Experiment on a wider variety of queries and databases
- Consider optimizations beyond Project-Join queries
- Experiment with other structural techniques, ie mini-buckets, clustering, treewidth approximation, etc.