

From Pebble Games to Query Optimization

Ben McMahan, Moshe Y. Vardi
Rice University

`mcmahanb@rice.edu, vardi@rice.edu`

September 18, 2004

What is a Query?

A query consists of operations on tables:

- Select (σ) \rightarrow Returns tuples that satisfy a given predicate.
- Project (π) \rightarrow Returns attributes listed.
- Join (\bowtie) \rightarrow Returns a filtered cross product of its arguments.
- Set operations \rightarrow union, intersect, and difference.

Most common queries are Select-Project-Join queries.

What is Query Optimization?

- Transform queries to logically equivalent queries.
- **Note:** Equivalent queries produce different running times:
 - $(r1 \bowtie r2) \bowtie \emptyset$ vs. $(\emptyset \bowtie r1) \bowtie r2$
- We call a particular method of execution a **plan**.
- Databases use cost-based optimization.

What is Cost-Based Optimization?

Cost-based optimization is a search technique that requires

- A **cost estimation** method for each plan, and
- A **search** algorithm.

The goal is to find an **accurate** cost estimation method and an **efficient** search algorithm to find a **low cost** plan.

PROBLEM: Scalability

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 $O(n2^{n-1})$.
- Exhaustive search is intractable.
- Local search is inefficient.

Where Might This Be a Problem?

Large number of joins appear in **machine generated** queries:

- Mediation systems,
- Complex views joined with other complex views.

Machine generated queries continually grow in use.

An Alternative Approach: Structural Heuristics

Structural Heuristics

- Optimize using 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 topological structure of the join graph.

Conjunctive-Query Evaluation

Conjunctive queries consist of queries expressible with Select-Project-Join.

Theorem: Given a conjunctive query Q and a database B , the following are equivalent:

- $Q(B)$ is true.
- There is a homomorphism $h : Q^D \rightarrow B$, where Q^D is the *canonical* database of query Q [CM77].

This homomorphism can be tested using the *existential pebble game*.

Existential k -Pebble Game

Let A and B be two relational structures. The **existential k -pebble game on A and B** consists of:

- Two players, the **Spoiler** and the **Duplicator**.
- Spoiler places or removes a pebble from an element of A .
- Duplicator tries to duplicate move on B .
- If $a_i \mapsto b_i$ for $1 \leq i \leq k$ is not a homomorphism, the **Spoiler wins**, otherwise the game continues.

Pebble Game Continued

There is a PTIME algorithm to decide whether the Spoiler or the Duplicator has a winning strategy for the existential k -pebble game.

Fact: If the Spoiler wins, there is no homomorphism.

Theorem: If the query's join graph has treewidth $k - 1$ and the Duplicator wins the existential k -pebble game, then there is a homomorphism [KV00].

Bringing It All Together

If we know the treewidth k of a query's join graph, we now have an algorithm in $O(n^k)$ to evaluate the query.

The **join graph** of a query creates

- A vertex for every attribute.
- An edge between attributes in the same relation.

We do not need to evaluate the pebble game, the k pebbles provides a bound on the size of intermediate results.

Rewriting the Query

Theorem: Queries with treewidth $k - 1$ are expressible in L^k [DKV02].

L^k are conjunctive queries with only k distinct variables.

New Approach: Look for a rewriting of the query that reduces the number of variables needed.

Problems

While finding out if a graph is treewidth k is linear, finding the treewidth of a graph is NP-hard.

Use an alternative approach, **Bucket Elimination** from Artificial Intelligence.

Produces directly an appropriate rewriting of the query.

Main Result: A heuristical bucket elimination approach provides an **exponential improvement** over greedy techniques.

Challenging Cost-based Optimizers

A simple setup to challenge cost-based optimizers:

- Small database – one table, two attributes, and six tuples.
- Large queries – hundreds of joins.
- Focused on Project-Join queries.
- Consider Boolean queries (output is empty or non-empty).

3-COLOR

We generate our queries from **3-COLOR** problems

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 $(u, v) \in E$ we have $c(u) \neq c(v)$.

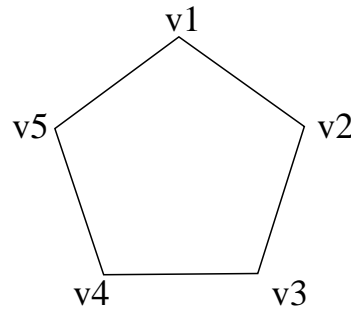
3-COLOR as a Query

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

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

Pentagon Example



The corresponding query of the pentagon above:

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

Random Queries

We generate random 3-COLOR graphs using two parameters:

- The **order** – number of vertices,
- The **density** – number of edges / order.
- For each edge two distinct vertices are picked uniformly.
- Edges are created, without repetition, until all edges have been generated.

Scaling

We are concerned with two type of scalability

- Density scaling – Fix order, increase density
 - Tests scalability over structural changes in the query.
 - Move from underconstrained to overconstrained instances.
- Order scaling – Fix density, increase order

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

Our Approach

Using PostgreSQL, for each graph,

- We construct an SQL query,
- Run the query,
- Gather results and both compilation and execution time.

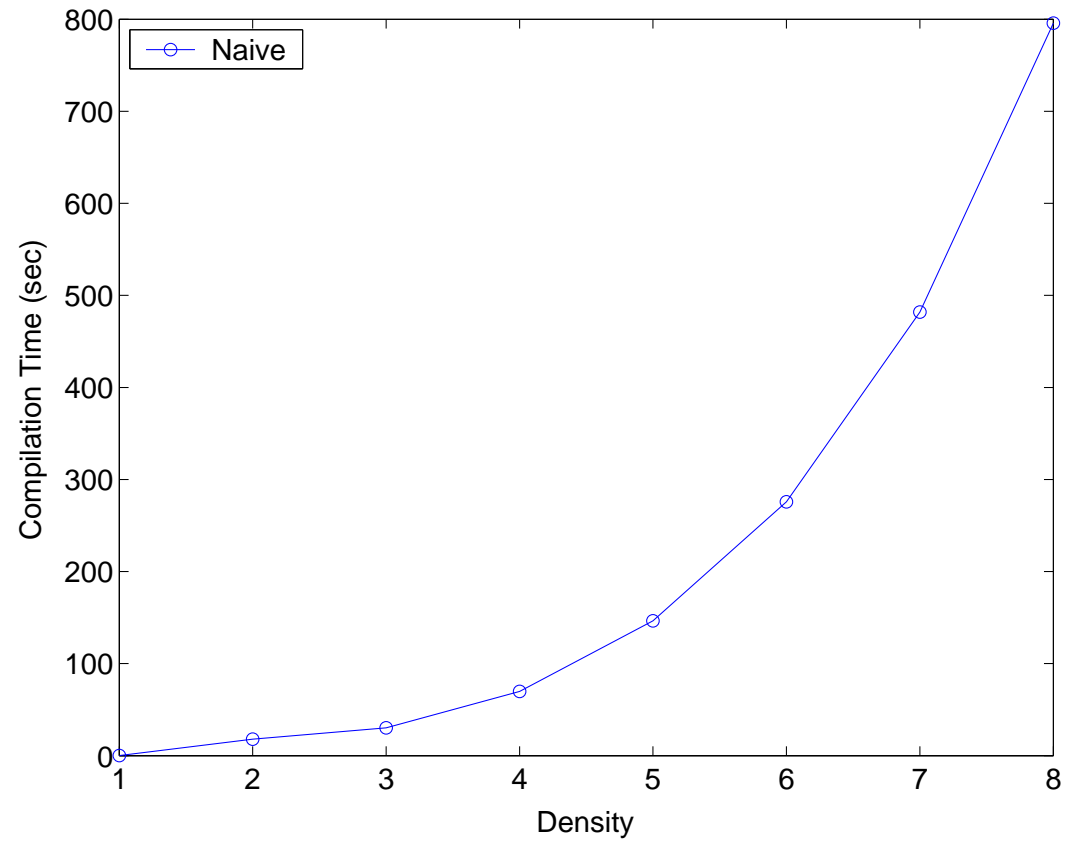
Naive Query

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);
```

Compile Time - Order 5



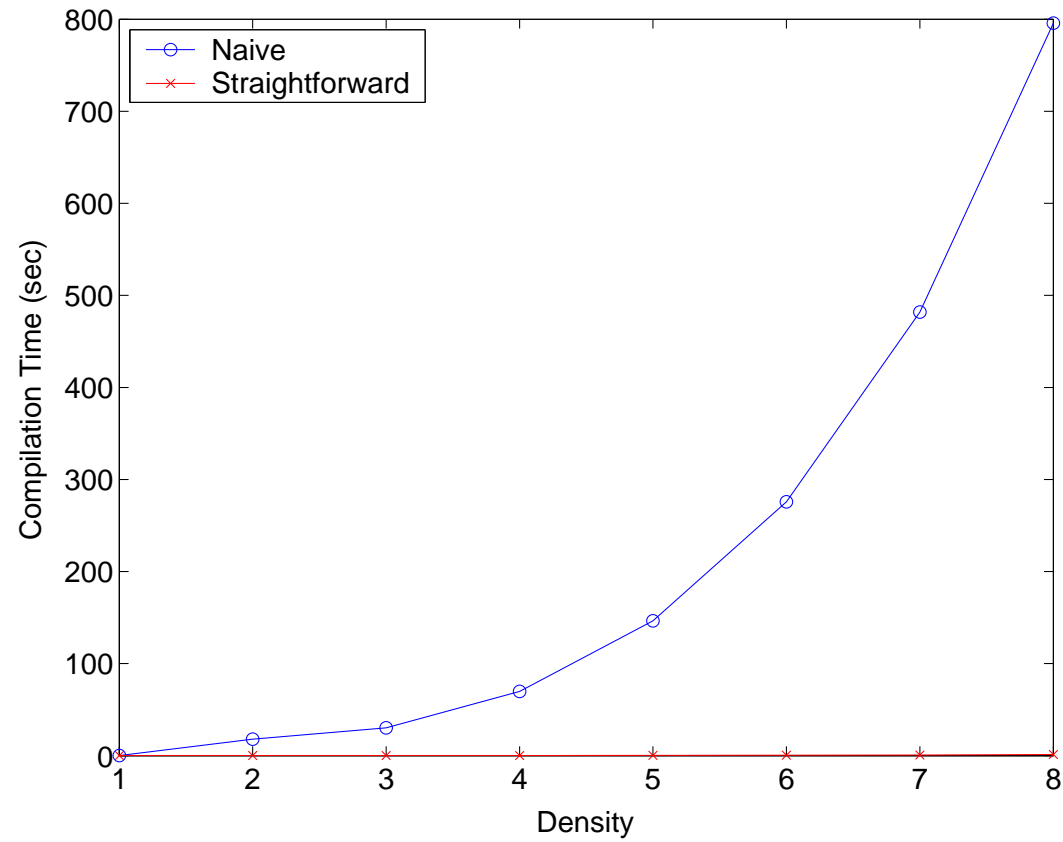
Straightforward Query

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)  
        ))))
```

Compile Time - Order 5



Naive vs. Straightforward

- NATURAL JOIN assumes equality on same names.
- Execution time the same as Naive.
- Compilation time decreased by **three orders of magnitude**.
 - Naive using local search, exhaustive search infeasible to use.
- Neither plan uses early projection!

What is Early Projection?

Consider the database with relations:

$$R_1 = (v_1, v_3) \text{ and } R_2 = (v_2, v_4)$$

Then the query $\pi_{v_1, v_2}(R_1 \bowtie_{v_1=v_2} R_2)$ could be rewritten as

$$(\pi_{v_1} R_1) \bowtie_{v_1=v_2} (\pi_{v_2} R_2).$$

- Neither naive nor straightforward plans used this.
- This was also true of DB2 and Oracle.
- Is early projection really not useful?

Early Projection

Our queries have the form $\pi_{\emptyset}(r_1 \bowtie \dots \bowtie r_m)$.

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

$$\pi_{\emptyset}(\pi_{livevars}(r_1 \bowtie \dots \bowtie r_q) \bowtie r_{q+1} \bowtie \dots \bowtie r_m).$$

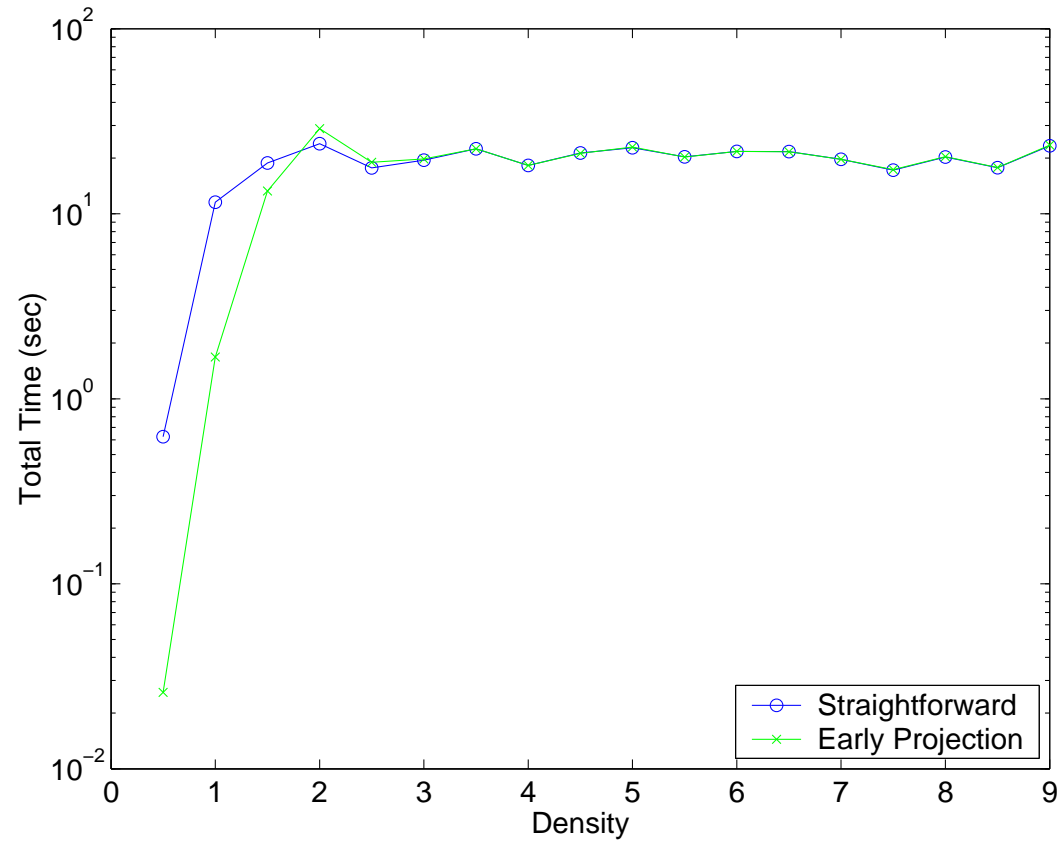
- livevars contains all the variables except v_j .
- v_j has been **projected early**.
- Arity of intermediate results has been reduced.

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 );
```

Early Projection Results



Reordering Relations

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

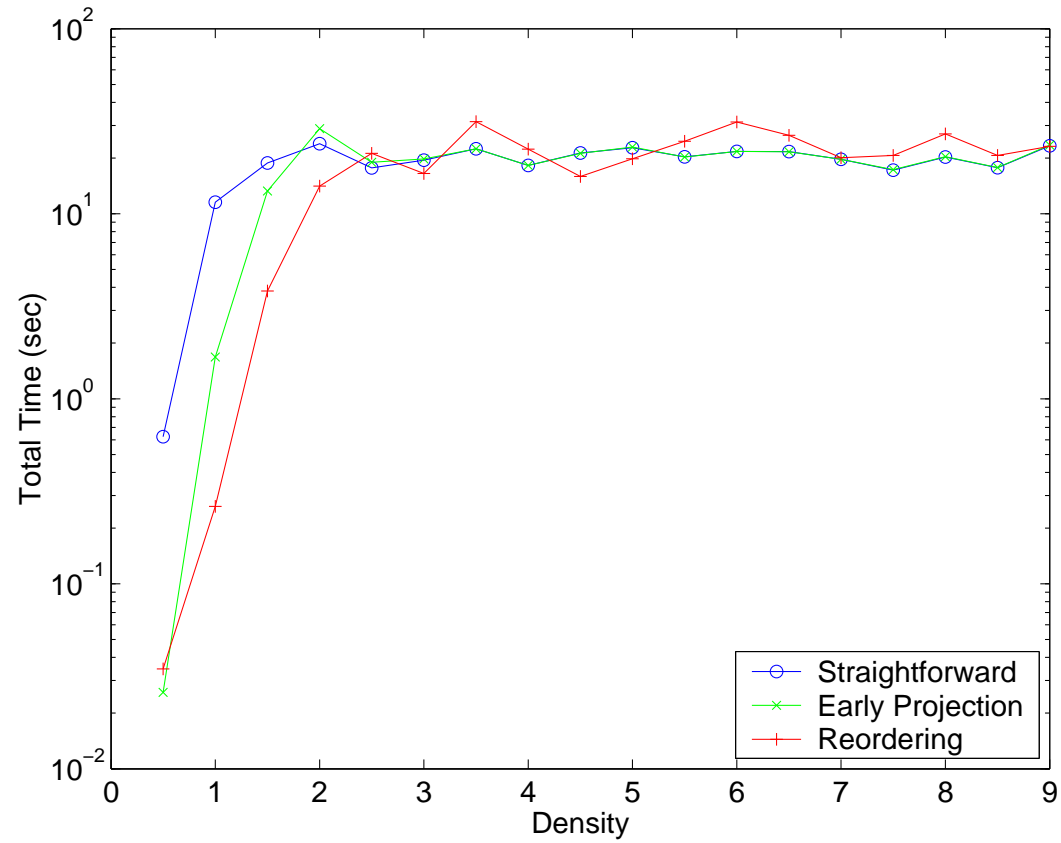
- Let v_1 be only in r_1 and r_m ,
- Then v_1 will not be projected early,
- But v_1 could be projected out after one join.

Greedy Heuristic

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

- Compute 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.

Reordering Results



From Greedy to Bucket Elimination

Early projection is an approach to minimizing the arity of the intermediate results.

The treewidth of the join graph is optimal with respect to arity under early projection [DKV02].

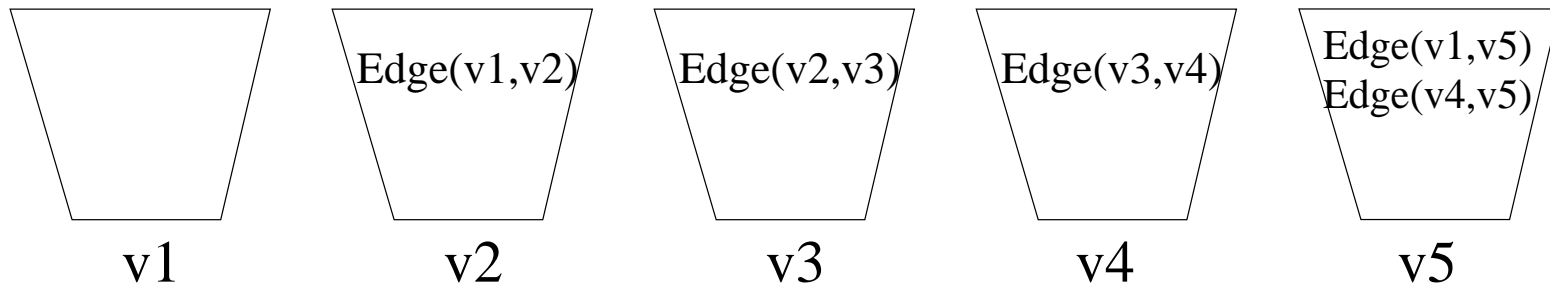
Use Bucket Elimination to approximate treewidth and minimize arity.

Given a correct variable order, Bucket Elimination will produce an optimal query.

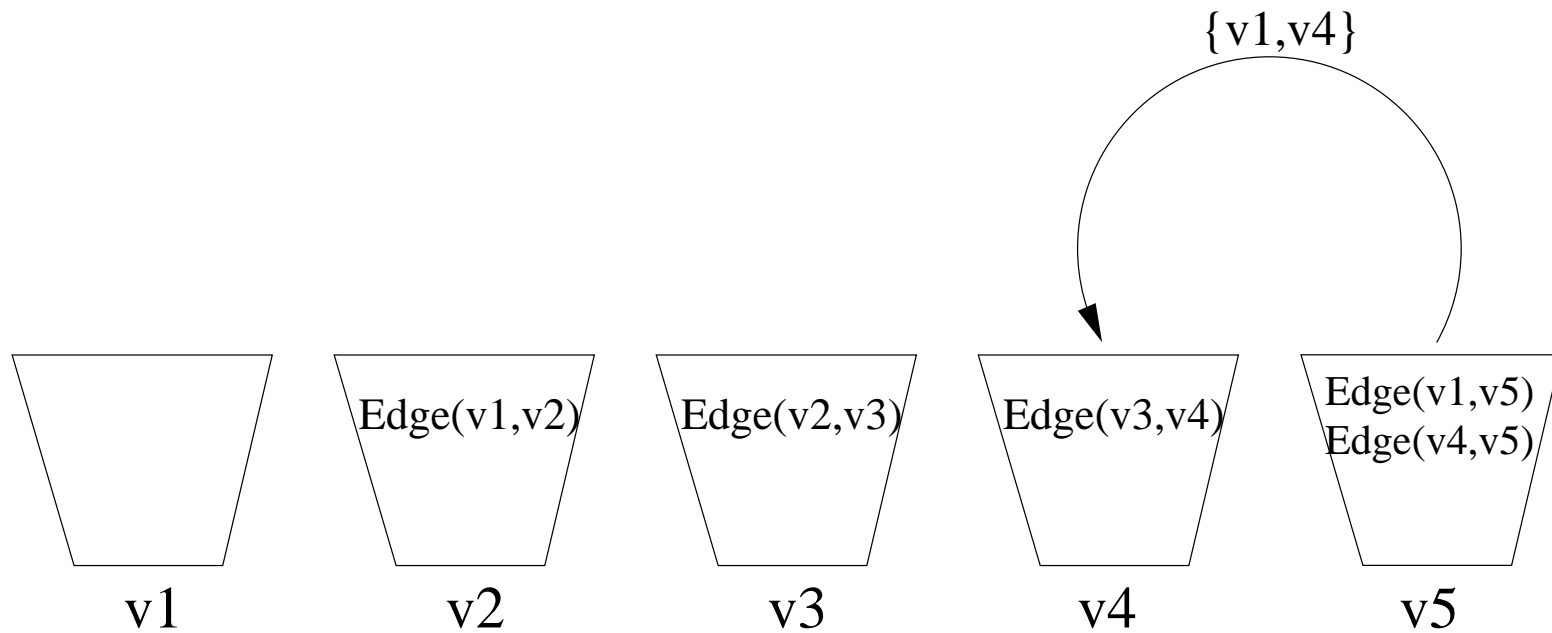
Bucket Elimination

- 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 appropriate bucket.

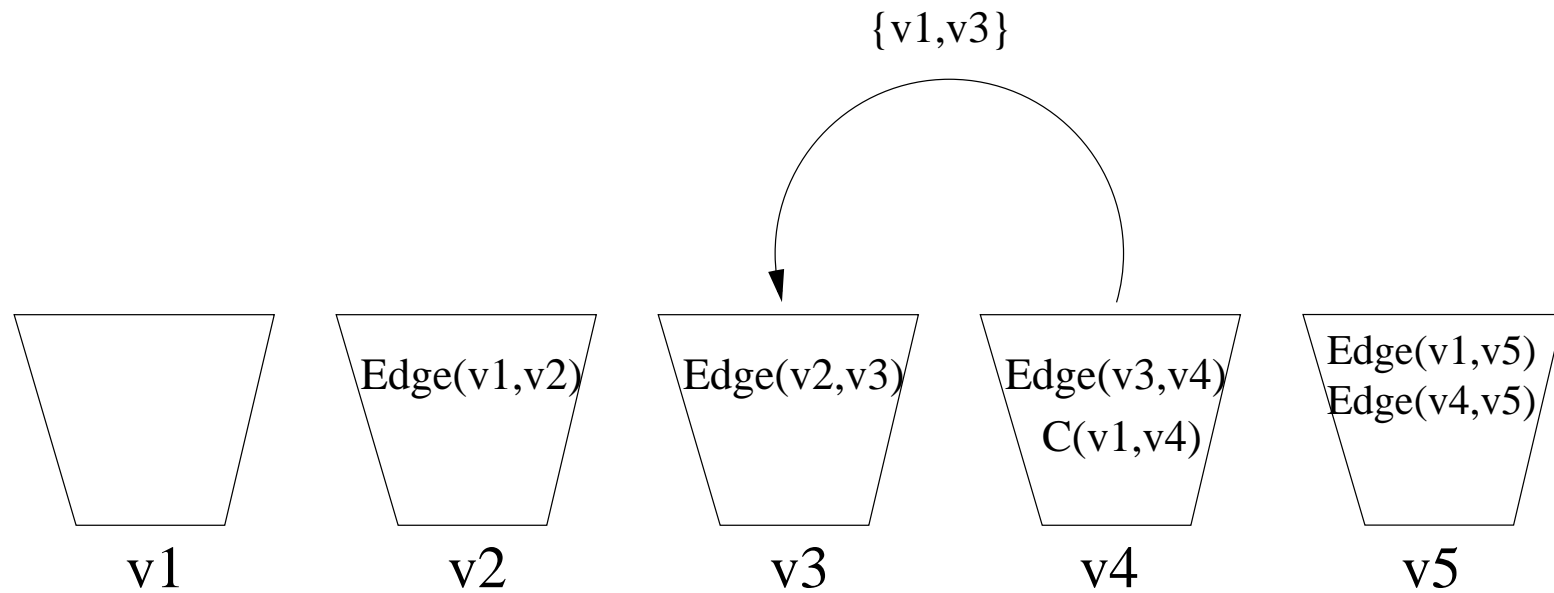
Bucket Elimination



Bucket Elimination after one step



Bucket Elimination after two steps



Maximum Cardinality Search

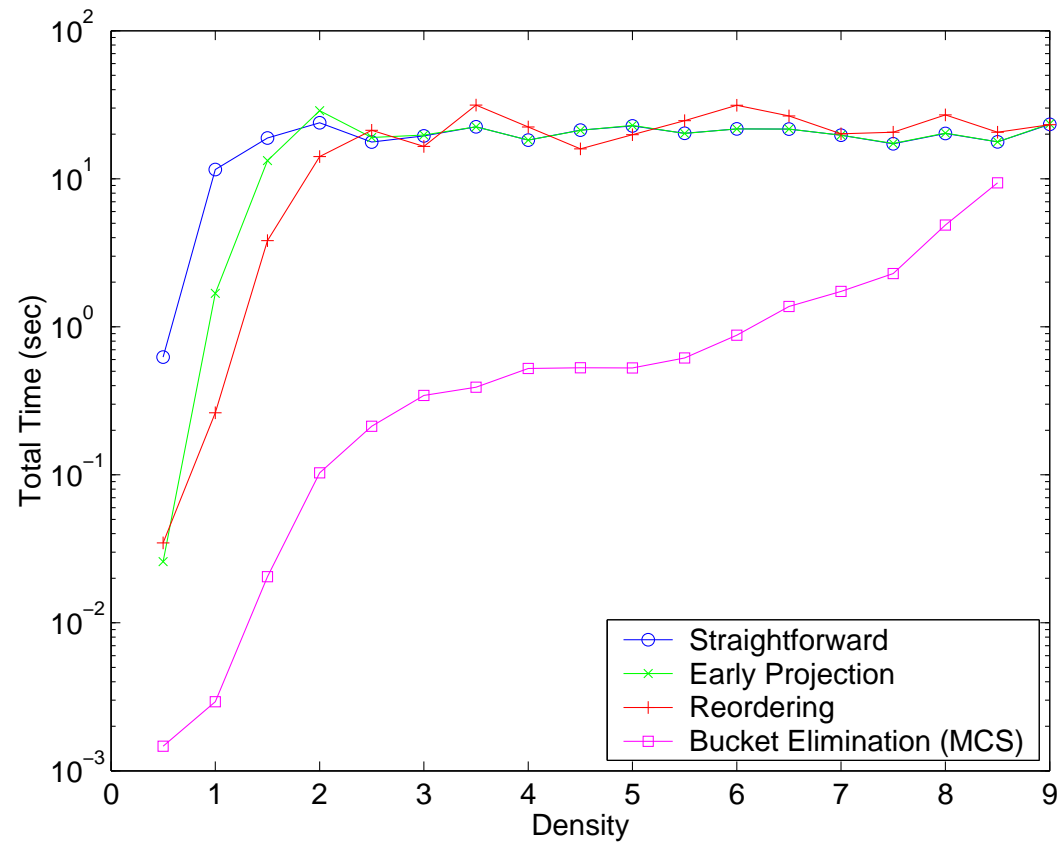
Finding optimal attribute order is NP-hard

Used the **Maximum Cardinality Search (MCS)** method:

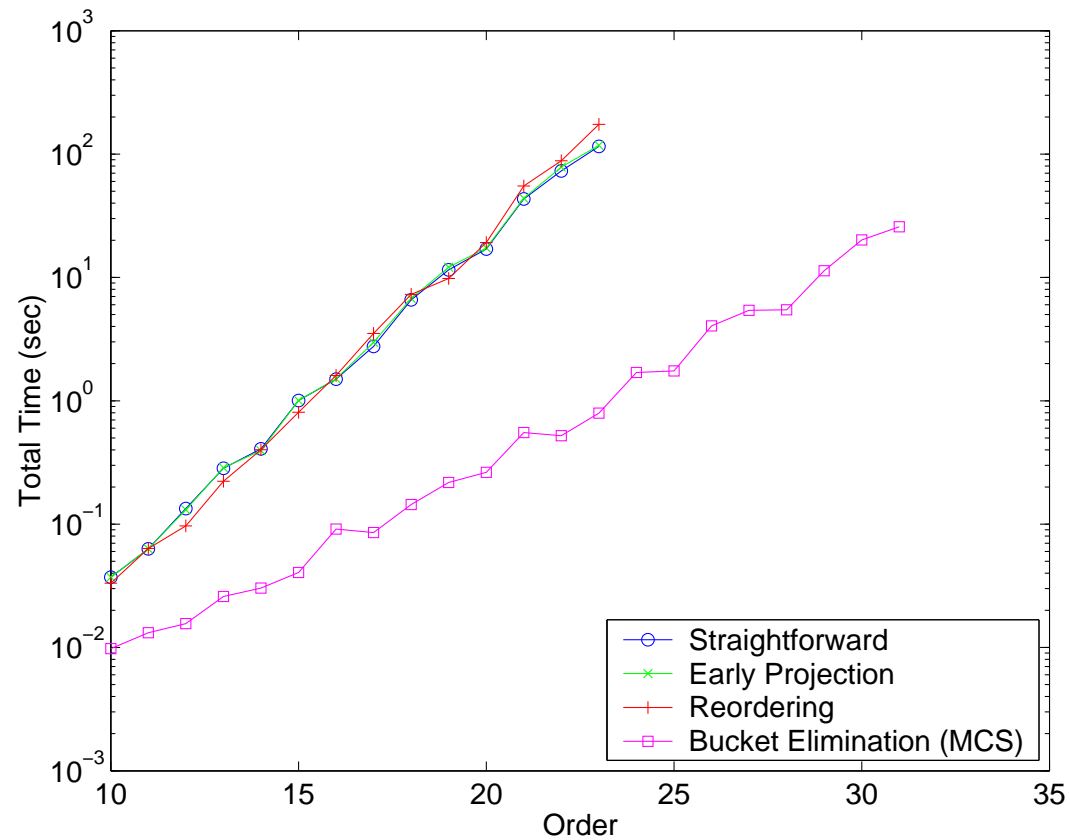
- Iterating over the join graph.
- Each iteration picks the attribute most connected to those already chosen.
- Ties broken arbitrarily.

MCS has been used successfully in constraint satisfaction.

Density Scaling - Order 20 - Logscale

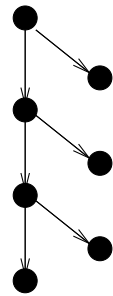


Order Scaling - Density 3.0 - Logscale

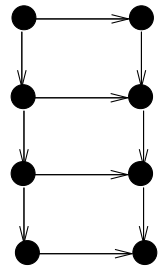


Structured Queries

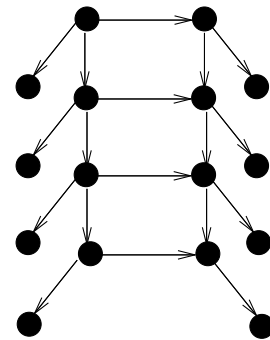
We also used structured queries



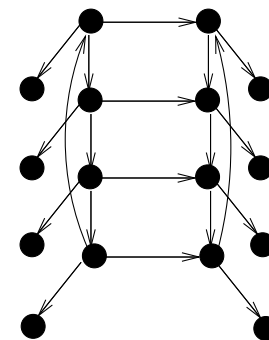
(a)



(b)



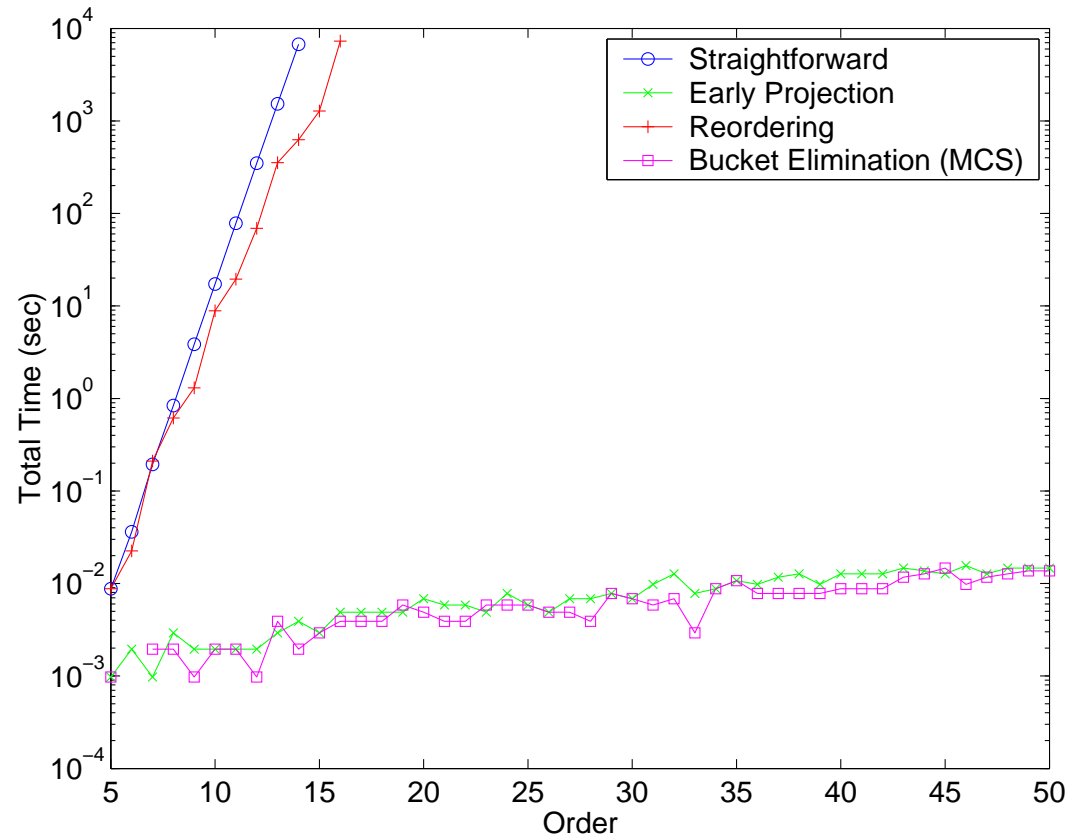
(c)



(d)

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

Augmented Path - Logscale



Conclusions

- Early projection, applied greedily, provides minimal improvement over straightforward approaches.
- Bucket elimination provides an **exponential improvement**.
- Structural heuristics can be used to optimize queries successfully.

Note that our techniques are applicable for general Project-Join queries.

Future Work

- Find a framework in which to combine cost-based and structural techniques, i.e., node and edge weighted graphs.
- Experiment on a wider variety of queries and databases.
- Consider optimizations beyond Project-Join queries.
- Experiment with other structural techniques, i.e., mini-buckets, clustering, hypertree decompositions, etc.