

Projection Pushing Revisited ^{*}

Benjamin J. McMahan^{**}, Guoqiang Pan^{***}, Patrick Porter[†], and Moshe Y. Vardi[‡]

Rice University

Abstract. The join operation, which combines tuples from multiple relations, is the most fundamental and, typically, the most expensive operation in database queries. The standard approach to join-query optimization is cost based, which requires developing a cost model, assigning an estimated cost to each query-processing plan, and searching in the space of all plans for a plan of minimal cost. Two other approaches can be found in the database-theory literature. The first approach, initially proposed by Chandra and Merlin, focused on minimizing the number of joins rather than on selecting an optimal join order. Unfortunately, this approach requires a homomorphism test, which itself is NP-complete, and has not been pursued in practical query processing. The second, more recent, approach focuses on structural properties of the query in order to find a project-join order that will minimize the size of intermediate results during query evaluation. For example, it is known that for Boolean project-join queries a project-join order can be found such that the arity of intermediate results is the treewidth of the join graph plus one.

In this paper we pursue the structural-optimization approach, motivated by its success in the context of constraint satisfaction. We chose a setup in which the cost-based approach is rather ineffective; we generate project-join queries with a large number of relations over databases with small relations. We show that a standard SQL planner (we use PostgreSQL) spends an exponential amount of time on generating plans for such queries, with rather dismal results in terms of performance. We then show how structural techniques, including projection pushing and join reordering, can yield exponential improvements in query execution time. Finally, we combine early projection and join reordering in an implementation of the bucket-elimination method from constraint satisfaction to obtain another exponential improvement.

1 Introduction

The join operation is the most fundamental and, typically, the most expensive operation in database queries. Indeed, most database queries can be expressed as select-project-join queries, combining joins with selections and projections. Choosing an optimal

^{*} Work supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, and EIA-0086264, and gifts from HP and Intel.

^{**} Address: Department of Computer Science, Rice University, Houston, TX 77005-1892, U.S.A.
Email: mcmahanb@rice.edu

^{***} Address: Department of Computer Science, Rice University, Houston, TX 77005-1892, U.S.A.
Email: gqpan@rice.edu

[†] Address: Scalable Software Email: Patrick.Porter@scalablessoftware.com

[‡] Address: Department of Computer Science, Rice University, Houston, TX 77005-1892, U.S.A.
Email: vardi@cs.rice.edu

plan, i.e., the particular order in which to perform the select, project, and join operations in the query, can have a drastic impact on query processing time and is therefore a key focus in query-processing research [32, 19].

The standard approach is that of *cost-based optimization* [22]. This approach requires the development of a cost model, based on database statistics such as relation size, block size, and selectivity, which enables assigning an estimated cost to each plan. The problem then reduces to searching the space of all plans for a plan of minimal cost. The search can be either exhaustive, for search spaces of limited size (cf. [4]), or incomplete, such as simulated annealing or other approaches (cf. [25]). In constructing candidate plans, one takes into account the fact that the join operation is associative and commutative, and that selections and projections commute with joins under certain conditions (cf. [34]). In particular, selections and projections can be “pushed” downwards, reducing the number of tuples and columns in intermediate relations [32]. Cost-based optimizations are effective when the search space is of manageable size and we have reasonable cost estimates, but it does not scale up well with query size (as our experiments indeed show).

Two other approaches can be found in the database-theory literature, but had little impact on query-optimization practice. The first approach, initially proposed by Chandra and Merlin in [8] and then explored further by Aho, Ullman, and Sagiv in [3, 2], focuses on minimizing the number of joins rather than on selecting a plan. Unfortunately, this approach generally requires a *homomorphism* test, which itself is NP-hard [20], and has not been pursued in practical query processing.

The second approach focuses on structural properties of the query in order to find a project-join order that will minimize the size of intermediate results during query evaluation. This idea, which appears first in [34], was first analytically studied for acyclic joins [35], where it was shown how to choose a project-join order in a way that establishes a linear bound on the size of intermediate results. More recently, this approach was extended to general project-join queries. As in [35], the focus is on choosing the project-join order in such a manner so as to polynomially bound the size of intermediate results [10, 21, 26, 11]. Specific attention was given in [26, 11] to *Boolean* project-join queries, in which all attributes are projected out (equivalently, such a query simply tests the nonemptiness of a join query). For example, [11] characterizes the minimal arity of intermediate relations when the project-join order is chosen in an optimal way and projections are applied as early as possible. This minimal arity is determined by the *join graph*, which consists of all attributes as nodes and all relation schemes in the join as cliques. It is shown in [11] that the minimal arity is the *treewidth* of the join graph plus one. The treewidth of a graph is a measure of how close this graph is to being a tree [16] (see formal definition in Section 5). The arity of intermediate relations is a good proxy for their size, since a constant-arity bound translates to a polynomial-size bound. This result reveals a theoretical limit on the effectiveness of projection pushing and join reordering in terms of the treewidth of the join graph. (Note that the focus in [28] is on projection pushing in recursive queries; the non-recursive case is not investigated.)

While acyclicity can be tested efficiently [31], finding the treewidth of a graph is NP-hard [5]. Thus, the results in [11] do not directly lead to a feasible way of finding an optimal project-join order, and so far the theory of projection pushing, as developed

in [10, 21, 26, 11], has not contributed to query optimization in practice. At the same time, the projection-pushing strategy has been applied to solve constraint-satisfaction problems in Artificial Intelligence with good experimental results [29, 30]. The input to a constraint-satisfaction problem consists of a set of variables, a set of possible values for the variables, and a set of constraints between the variables; the question is to determine whether there is an assignment of values to the variables that satisfies the given constraints. The study of constraint satisfaction occupies a prominent place in Artificial Intelligence, because many problems that arise in different areas can be modeled as constraint-satisfaction problems in a natural way; these areas include Boolean satisfiability, temporal reasoning, belief maintenance, machine vision, and scheduling [14]. A general method for projection pushing in the context of constraint satisfaction is the *bucket-elimination* method [15, 14]. Since evaluating Boolean project-join queries is essentially the same as solving constraint-satisfaction problems [26], we attempt in this paper to apply the bucket-elimination approach to approximate the optimal project-join order (i.e., the order that bounds the arity of the intermediate results by the treewidth plus one).

In order to focus solely on projection pushing in project-join expressions, we choose an experimental setup in which the cost-based approach is rather ineffective. To start, we generate project-join queries with a large number (up to and over 100) of relations. Such expressions are common in mediator-based systems [36]. They challenge cost-based planners, because of the exceedingly large size of the search space, leading to unacceptably long query compile time. Furthermore, to factor out the influence of cost information we consider small databases, which fit in main memory and where cost information is essentially irrelevant. (Such databases arise naturally in query containment and join minimization, where the query itself is viewed as a database [8]. For a survey of recent applications of query containment see [23].) We show experimentally that a standard SQL planner (we use PostgreSQL) spends an exponential amount of time on generating plans for such queries, with rather dismal results in terms of performance and without taking advantage of projection pushing (and neither do the SQL planners of DB2 and Oracle, despite the widely held belief that projection pushing is a standard query-optimization technique).

Our experimental test suite consists of a variety of project-join queries. We take advantage of the correspondence between constraint satisfaction and project-join queries [26] to generate queries and databases corresponding to instances of 3-COLOR problems [20]. Our main focus in this paper is to study the scalability of various projection-pushing methods. Thus, our interest is in comparing the performance of different optimization techniques when the size of the queries is increased. We considered both random queries as well as a variety of queries with specific structures, such as “augmented paths”, “ladders”, “augmented ladders”, and “augmented circular ladders” [27]. To study the effectiveness of the bucket-elimination approach, we start with a straightforward plan that joins the relations in the order in which they are listed, without applying projection pushing. We then proceed to apply projection pushing and join reordering in a greedy fashion. We demonstrate experimentally that this yields exponential improvement in query execution time over the straightforward approach. Finally, we combine projection pushing and join reordering in an implementation of the bucket-

elimination method. We first prove that this method is optimal for general project-join queries with respect to intermediate-result arity, provided the “right” order of “buckets” is used (this was previously known only for Boolean project-join queries [15, 17, 26, 11]). Since finding such an order is NP-hard [5], we use the “maximum cardinality” order of [31], which is often used to approximate the optimal order [7, 29, 30]. We demonstrate experimentally that this approach yields an exponential improvement over the greedy approach for the complete range of input queries in our study. This shows that applying bucket elimination is highly effective even when applied heuristically and it significantly dominates greedy heuristics, without incurring the excessive cost of searching large plan spaces.

The outline of the paper is as follows. Section 2 describes the experimental setup. We then describe a naive and straightforward approach in Section 3, a greedy heuristic approach in Section 4, and the bucket-elimination approach in Section 5. We report on our scalability experiments in Section 6, and conclude with a discussion in Section 7.

2 Experimental Setup

Our goal is to study join-query optimization in a setting in which the traditional cost-based approach is ineffective, since we are interested in using the structure of the query to drive the optimization. Thus, we focus on project-join queries with a large number of relations over a very small database, which not only easily fits in main memory, but also where index or selectivity information is rather useless. First, to neutralize the effect of query-result size we consider Boolean queries in which all attributes are projected out, so the final result consists essentially of one bit (empty result vs. nonempty result). Then we also consider queries where a fraction of attributes are not projected out, to simulate more closely typical database usage.

We work with project-join queries, also known as *conjunctive queries* (conjunctive queries are usually defined as positive, existential conjunctive first-order formulas [1]). Formally, an n -ary project-join query is a query definable by the project-join fragment of relational algebra; that is, by an expression of the form $\pi_{x_1, \dots, x_n}(R_1 \bowtie \dots \bowtie R_m)$, where the R_j s are relations and x_1, \dots, x_n are the free variables (we use the terms “variables” and “attributes” interchangeably). Initially, we focus on Boolean project-join queries, in which there are no free variables. We emulate Boolean queries by including only a single variable in the projection (i.e., $n = 1$). Later we consider non-Boolean queries where there are a fixed fraction of free variables by listing them in the projection.

In our experiments, we generate queries with the desired characteristics by translating graph instances of 3-COLOR into project-join queries, cf. [8]. This translation yields queries over a single binary relation with six tuples. It is important to note, however, that our algorithms do not rely on the special structure of the queries that we get from 3-COLOR instances (i.e., bounded arity of relations and bounded domain size)—they are applicable to project-join queries in general. An instance of 3-COLOR is a graph $G = (V, E)$ and a set of colors $C = \{1, 2, 3\}$, where $|V| = n$ and $|E| = m$. For each edge $(u, v) \in E$, there are six possible colorings of (u, v) to satisfy the requirement of no monochromatic edges. We define the relation *edge* as containing all six tuples corresponding to all pairs of distinct colors. The query Q_G is then expressed as

the project-join expression $\pi_{v_1} \bowtie_{(v_i, v_j) \in E} \text{edge}(v_i, v_j)$. This query returns a nonempty result over the *edge* relation iff G is 3-colorable [8].

We note that our queries have the following features (see our concluding remarks):

- Projecting out a column from our relation yields a relation with all possible tuples. Thus, in our setting, *semijoins*, as in the Wong-Youssefi algorithm [34], are useless. This enables us to focus solely on ordering joins and projections.
- Since we only have one relation of fixed arity, differences between the various notions of width, such as treewidth, query width, and hypertree widths are minimized [10, 21], enabling us to focus in this paper on treewidth.

To generate a range of queries with varying structural properties, we start by generating random graph instances. For a fixed number n of vertices and a fixed number m of edges, instances are generated uniformly. An edge is generated by choosing uniformly at random two distinct vertices. Edges are generated (without repetition) until the right number of edges is arrived at. We measure the performance of our algorithms by scaling up the size of the queries (note that the database here is fixed). We focus on two types of scalability. First, we keep the *order* (i.e. the number of vertices) fixed, while scaling up the *density*, which is the ratio m/n of edges to vertices. Second, we keep the density fixed while scaling up the order. Clearly, a low density suggests that the instance is underconstrained, and therefore is likely to be 3-colorable, while a high density suggests that the instance is overconstrained and is unlikely to be 3-colorable. Thus, density scaling yields a spectrum of problems, going from underconstrained to overconstrained. We studied orders between 10 and 35 and densities between 0.5 and 8.0.

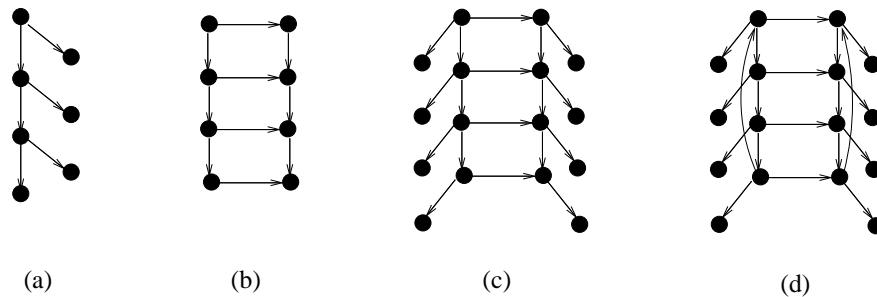


Fig. 1. Augmented path, ladder, augmented ladder, and augmented circular ladder

We also consider non-random graph instances for 3-COLOR. These queries, suggested in [27], have specific structures. An *augmented path* (Figure 1a) is a path of length n , where for each vertex on the path a dangling edge extends out of the vertex. A *ladder* (Figure 1b) is a ladder with n rungs. An *augmented ladder* (Figure 1c) is a ladder where every vertex has an additional dangling edge, and an *augmented circular ladder* (Figure 1d) is an augmented ladder where the top and bottom vertices are connected together with an edge. For these instances only the order is scaled; we used orders from 5 to 50.

All experiments were performed on the Rice Tersascale Cluster¹, which is a Linux cluster of Itanium II processors with 4GB of memory each, using PostgreSQL² 7.2.1. For each run we measured the time it took to generate the query, the time the PostgreSQL Planner took to optimize the query, and the execution time needed to actually run the query. For lack of space we report only median running times. Using command-line parameters we selected hash joins to be the default, as hash joins proved most efficient in our setting.

3 Naive and Straightforward Approaches

Given a conjunctive query $\varphi = \pi_{v_1} \bowtie_{(v_i, v_j) \in E} \text{edge}(v_i, v_j)$, we first use a *naive* translation of φ into SQL:

```
SELECT DISTINCT  $e_1.u_1$ 
FROM  $\text{edge } e_1(u_1, w_1), \dots, \text{edge } e_m(u_m, w_m)$ 
WHERE  $\bigwedge_{j=1}^m (e_j.u_j = e_{p(u_j)}.u_j \text{ AND } e_j.w_j = e_{p(w_j)}.w_j)$ 
```

As SQL does not explicitly allow us to express Boolean queries, we instead put a single vertex in the SELECT section. The FROM section simply enumerates all the atoms in the query, referring to them as e_1, \dots, e_m and renames the columns to match the vertices of the query. The WHERE section enforces equality of different occurrences of the same vertex. More precisely, we enforce equality of each occurrence to the first occurrence of the same vertex; $p(v_i)$ points to the first occurrence of the vertex v_i .

We ran these queries for instances of order 5 and integral densities from 1 to 8 (the database engine could not handle larger orders with the naive approach). The PostgreSQL Planner found the naive queries exceedingly difficult to compile; compile time was four orders of magnitude longer than execution time. Furthermore, compile time scaled exponentially with the density as shown in Figure 2. We used the PostgreSQL Planner’s genetic algorithm option to search for a query plan, as an exhaustive search for our queries was infeasible. This algorithm proved to be quite slow as well as ineffective for our queries. The plans generated by the Planner showed that it does not utilize at all projection pushing; it simply chooses some join order.

In an attempt to get around the Planner’s ineffectiveness, we implemented a *straightforward* approach. We explicitly list the joins in the FROM section of the query, instead of using equalities in the WHERE section as in the naive approach.

```
SELECT DISTINCT  $e_1.u_1$ 
FROM  $\text{edge } e_1(u_1, w_1) \text{ JOIN } \dots \text{ JOIN } \text{edge } e_m(u_m, w_m)$ 
ON  $(e_1.u_1 = e_{p(u_1)}.u_1 \text{ AND } e_1.w_1 = e_{p(w_1)}.w_1)$  ON  $\dots$  ON  $(e_m.u_m = e_{p(u_m)}.u_m \text{ AND } e_m.w_m = e_{p(w_m)}.w_m)$ 
```

Parentheses forces the evaluation to proceed from e_1 to e_2 and onwards (i.e., $(\dots (e_1 \bowtie e_2) \dots \bowtie e_m)$). We omit parentheses here for sake of readability.

¹ <http://www.citi.rice.edu/rtc/>

² <http://www.postgresql.org/>

The order in which the relations are listed then becomes the order that the database engine evaluates the query. This effectively limits what the Planner can do and therefore drastically decreases compile time. As is shown in Figure 2, compile time still scales exponentially with density, but more gracefully than the compile time for the naive approach.

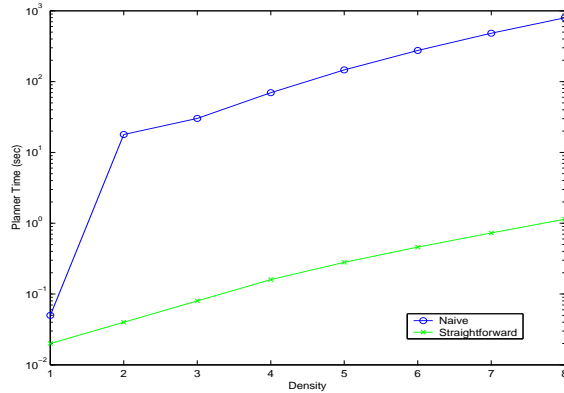


Fig. 2. Naive and straightforward approaches: density scaling of compile time, 3-SAT, 5 variables, logscale

We note that the straightforward approach also does not take advantage of projection pushing. We found query execution time for the naive and straightforward approaches to be essentially identical; the join order chosen by the genetic algorithm is apparently no better than the straightforward order.

In the rest of the paper we show how we can take advantage of projection pushing and join reordering to improve query execution time dramatically. As a side benefit, since we use subqueries to enforce a particular join and projection order, compile time becomes rather negligible, which is why we do not report it.

4 Projection Pushing and Join Reordering

The conjunctive queries we are considering have the form $\pi_{v_1}(e_1 \bowtie e_2 \bowtie \dots \bowtie e_m)$. If v_j does not appear in the relations e_{k+1}, \dots, e_m , then we can rewrite the formula into an equivalent one:

$$\pi_{v_1}(\pi_{livevars}(e_1 \bowtie \dots \bowtie e_k) \bowtie \dots \bowtie e_m)$$

where *livevars* are all the variables in the scope minus v_j . This means we can write a query to join the relations e_1, \dots, e_k , project out v_j , and join the result with relations e_{k+1}, \dots, e_m . We then say that the projection of v_j has been pushed in and v_j has been *projected early*. The hope is that early projection would reduce the size of intermediate results by reducing their arity, making further joins less expensive, and thus reducing

the execution time of the query. Note that the reduction in size of intermediate results has to offset the overhead of creating a copy of the projected relations.

We implemented early projection in SQL using subqueries. The subformula found in the scope of each nested existential quantifier is itself a conjunctive query, therefore each nested existential quantifier becomes a subquery. Suppose that k and j above are minimal. Then the SQL query can be rewritten as:

```
SELECT DISTINCT  $e_m.u_m$ 
FROM  $edge\ e_m(u_m, w_m)$  JOIN ... JOIN  $edge\ e_{k+1}(u_{k+1}, w_{k+1})$  JOIN ( subquery $_k$ )
AS  $t_k$ 
ON ( $e_{k+1}.u_{k+1} = e_{p(u_{k+1})}.u_{p(u_{k+1})}$  AND  $e_{k+1}.w_{k+1} = e_{p(w_{k+1})}.w_{p(w_{k+1})}$ ) ON ...
ON ( $e_m.u_m = e_{p(u_m)}.u_{p(u_m)}$  AND  $e_m.w_m = e_{p(w_m)}.w_{p(w_m)}$ )
```

where subquery $_k$ is obtained by translating the subformula $\pi_{livevars}(e_1 \bowtie \dots \bowtie e_k)$ into SQL according to the straightforward approach (with $p(v_l)$ updated to point toward t_k , when appropriate, for occurrences of v_l in e_{k+1}, \dots, e_m). The only difference between the subquery and the full query is the SELECT section. In the subquery the SELECT section contains all *live* variables within its scope, i.e. all variables except for v_j . Finally, we proceed recursively to apply early projection to the join of e_{k+1}, \dots, e_m .

The early projection method processes the relations of the query in a linear fashion. Since the goal of early projection is to project variables as soon as possible, reordering the relations may enable us to project early more aggressively. For example, if a variable v_j appears only in the first and the last relation, early projection will not quantify v_j out. But had the relations been processed in a different order, v_j could have been projected out very early. In general, instead of processing the relations in the order e_1, \dots, e_m , we can apply a permutation ρ and process the relations in the order $e_{\rho(1)}, \dots, e_{\rho(m)}$. The permutation ρ should be chosen so as to minimize the number of live variables in the intermediate relations. This observation was first made in the context of symbolic model checking, cf. [24]. Finding an optimal permutation for the order of the relations is a hard problem in and of itself. So we have implemented a greedy approach, searching at each step for an atom that would result in the maximum number of variables to be projected early. The algorithm incrementally computes an atom order. At each step, the algorithm searches for an atom with the maximum number of variables that occur only once in the remaining atoms. If there is a tie, the algorithm chooses the atom that shares the least variables with the remaining atoms. Further ties are broken randomly. Once the permutation ρ is computed, we construct the same SQL query as before, but this time with the order suggested by ρ . We then call this method *reordering*.

5 Bucket Elimination

The optimizations applied in Section 4 correspond to a particular rewriting of the original conjunctive query according to the algebraic laws of the relational algebra [32]. By using projection pushing and join reordering, we have attempted to reduce the arity of intermediate relations. It is natural to ask what the limit of this technique is, that is, if we consider all possible join orders and apply projection pushing aggressively, what is the minimal upper bound on the arity of intermediate relations?

Consider a project-join query $Q = \pi_{x_1, \dots, x_n}(R_1 \bowtie \dots \bowtie R_m)$ over the set $\mathcal{R} = \{R_j | 1 \leq j \leq m\}$ of relations, where A is the set of attributes in \mathcal{R} , and the target schema $S_Q = \{x_1, \dots, x_n\}$ is a subset of A . A *join-expression tree* of Q can be defined as a tuple $J_Q = (T = (V_Q, E_Q, v_0), L_w, L_p)$ where T is a tree, with nodes V_Q and edges E_Q , rooted at v_0 , and both $L_w : V_Q \rightarrow 2^A$ and $L_p : V_Q \rightarrow 2^A$ label the nodes of T with sets of attributes. For each node $v \in V_Q$, $L_w(v)$ is called v 's *working label* and $L_p(v)$ is called v 's *projected label*. For every leaf node $u \in V_Q$ there is some $R_j \in \mathcal{R}$ such that $L_w(u) = R_j$. For every nonleaf node $v \in V_Q$, we define $L_w(v) = \bigcup_{\{x | (v,x) \in E_Q\}} L_p(x)$ as the union of the projected labels of its children. The projected label $L_p(u)$ of a node u is the subset of $L_w(u)$ that consists of all $a \in L_w(u)$ that appear outside the subtree of T rooted at u . All other attributes are said to be *unnecessary* for u . Intuitively, the join-expression tree describes an evaluation order for the join, where the joins are evaluated bottom up and projection is applied as early as possible for that particular evaluation order. The *width* of the join-expression tree J_Q is defined as $\max_{v \in V_Q} |L_w(v)|$, the maximum size of the working label. The *join width* of Q is the width over all possible join-expression trees of Q .

To understand the power of join reordering and projection pushing, we wish to characterize the join width of project-join queries. We now describe such a characterization in terms of the *join graph* of Q . In the join graph $G_Q = (V, E)$, the node set V is the set A of attributes, and the edge set E consists of all pairs (x, y) of attributes that co-occur in some relation $R_j \in \mathcal{R}$. Thus, each relation $R_j \in \mathcal{R}$ yields a clique over its attributes in G_Q . In addition, we add an edge (x, y) for every pair of attributes in the schema S_Q . The important parameter of the join graph is its *treewidth* [16].

Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair (T, X) , where $T = (I, F)$ is a tree with node set I and edge set F , and $X = \{X_i : i \in I\}$ is a family of subsets of V , one for each node of T , such that (1) $\bigcup_{i \in I} X_i = V$, (2) for every edge $(v, w) \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and (3) for all $i, j, k \in I$, if j is on the path from i to k in T , then $X_i \cap X_k \subseteq X_j$. The *width* of a tree decomposition is $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G , denoted by $tw(G)$, is the minimum width over all possible tree decompositions of G . For each fixed $k > 0$, there is a linear-time algorithm that tests whether a given graph G has treewidth k . The algorithm actually constructs a tree decomposition of G of width k [16].

We can now characterize the join width of project-join queries:

Theorem 1. *The join width of a project-join query Q is $tw(G_Q) + 1$.*

Proof Sketch: We first show that the join width of Q provides a bound for $tw(G_Q) + 1$. Given a join-expression tree J_Q of Q with width k , we construct a tree decomposition of G_Q of width $k - 1$. Intuitively, we just drop all the projected labels and use the working label as the tree decomposition labeling function. Algorithm 1 shows this conversion.

Lemma 1. *Given a project-join query Q and 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 $k - 1$.*

Proof Sketch:

In the other direction, we can go from tree decompositions to join-expression trees. First the join graph G_Q is constructed and a tree decomposition of width k for this graph

Algorithm 1 Join-Expression-Tree-to-Tree-Decomposition(Q, J_Q)

Require: Project-join query Q and join-expression tree $J_Q = (T = (V_Q, E_Q, v_0), L_w, L_p)$

Ensure: A tree decomposition $T = ((I, F), X)$

- 1: $I = V_Q$ {the nodes of the tree}
 - 2: $F = E_Q$ {the edges of the tree}
 - 3: $X = L_w$ {the labeling function}
 - 4: **RETURN** $T_{J_Q} = ((I, F), X)$
-

is constructed. Once we have a tree decomposition, we simplify it using Algorithm 2 to have only the nodes needed for the join-expression tree without increasing the width of the tree decomposition. In other words, the leaves of the simplified tree decomposition each correspond to a relation in $\mathcal{R} \cup \mathcal{S}_Q$, and the nodes between these leaves still maintain all the tree-decomposition properties.

Algorithm 2 Mark-and-Sweep($(T = (I, F), X), Q$)

Require: A tree decomposition $(T = (I, F), X)$ and a project-join query Q

Ensure: A simplified tree decomposition $(T' = (I', F'), X')$ and a mapping $r : \mathcal{R} \rightarrow I'$

- 1: **for** every relation $R_j \in \mathcal{R} \cup \{R_T\}$ **do**
 - 2: Find a node $i \in I$ such that $R_j \subseteq X_i$
 - 3: Mark R_j in X_i
 - 4: $r[R_j] = i$ {remember this is the node corresponding to the relation}
 - 5: **end for**
 - 6: **for** every pair of nodes $i, j \in I$ **do**
 - 7: **for** every node k along the path from i to j in T **do**
 - 8: Mark the subset of X_k where $\{x \mid x \in X_k, x \text{ is marked in } X_i, \text{ and } x \text{ is marked in } X_j\}$
 - 9: **end for**
 - 10: **end for**
 - 11: **for** every node $i \in I$ **do**
 - 12: Delete all unmarked labels in X_i .
 - 13: **if** $X_i == \emptyset$ **then**
 - 14: Delete i {If the label is empty, delete the node and corresponding edges}
 - 15: **for** all edges $e \in F$ containing i **do**
 - 16: Delete e
 - 17: **end for**
 - 18: **end if**
 - 19: **end for**
 - 20: **RETURN** $(T = (I, F), X), r$
-

Lemma 2. *Given a project-join query Q , its join graph G_Q , and a tree decomposition $(T = (I, F), X)$ of G_Q of width k , there is a simplified tree decomposition $(T' = (I', F'), X')$ of G_Q of width k such that every leaf node of T' has a label containing an R_j for some $R_j \in \mathcal{R}$.*

Once we have a simplified tree decomposition, Algorithm 3 shows how to convert it into a join-expression tree.

Algorithm 3 Tree-Decomposition-to-Join-Expression-Tree(Q, G_Q, T_{J_Q})

Require: A conjunctive query Q , iteration graph $G_{\mathcal{R}}$, and tree decomposition of G_Q , $T_{J_Q} = ((I, F), X)$

Ensure: A join-expression tree $J_Q = (T = (V_Q, E_Q, v_0), L_w, L_p)$

```

1:  $(T' = (I', F'), X'), r = \text{Mark-and-Sweep}(T_{J_Q}, Q)$ 
2:  $V_Q = I'$  {the nodes of the join-expression tree}
3:  $E_Q = F'$  {the edges of the join-expression tree}
4:  $v_0 = r[R_T]$  {set the root to be a superset of the target relation}
5: for every relation  $R_j \in \mathcal{R}$  do
6:    $z = \text{newnode}(R_j)$ 
7:    $V_Q = V_Q \cup z$  {create a new node for each relation, these will be the leaf nodes of the
   join-expression tree}
8:    $E_Q = E_Q \cup (r[R_j], z)$  {add an edge connecting the relational node to the tree decompo-
   sition}
9: end for
10: for every node  $i \in V_Q$  do {Set up working labels}
11:   if  $i$  is a leaf node introduced for relation  $R_j \in \mathcal{R}$  then
12:      $L_w(i) = R_j$ 
13:   else
14:      $L_w(i) = X'_i$ 
15:   end if
16: end for
17: for every node  $i \in V_Q$  do {Set up projected labels}
18:   if  $i$  is a leaf node introduced for relation  $R_j \in \mathcal{R}$  then
19:      $L_p(i) = R_j$ 
20:   else if  $i \neq v_0$  then
21:      $L_p(i) = L_w(i) \cap L_w(\text{parent}(i))$ 
22:   else
23:      $L_p(i) = R_T$ 
24:   end if
25: end for
26: RETURN  $((V_Q, E_Q, v_0), L_w, L_p)$ 

```

Lemma 3. *Given a project-join query Q , a join graph G_Q , and a simplified tree decomposition of G_Q of treewidth k , there is a join-expression tree of Q with join width $k + 1$.*

■

Theorem 1 offers a graph-theoretic characterization of the power of projection pushing and join reordering. The theorem extends results in [11] regarding rewriting of Boolean conjunctive queries, expressed in the syntax of first-order logic. That work explores rewrite rules whose purpose is to rewrite the original query Q into a first-order

formula using a smaller number of variables. Suppose we succeeded to rewrite Q into a formula of L^k , which is the fragment of first-order logic with k variables, containing all atomic formulas in these k variables and closed only under conjunction and existential quantification over these variables. We then know that it is possible to evaluate Q so that all intermediate relations are of width at most k , yielding a polynomial upper bound on query execution time [33]. Given a Boolean conjunctive query Q , we'd like to characterize the minimal k such that Q can be rewritten into L^k , since this would describe the limit of variable-minimization optimization. It is shown in [11] that if k is a positive integer and Q a Boolean conjunctive query, then the join graph of Q has treewidth $k - 1$ iff there is an L^k -sentence ψ that is a rewriting of Q . Theorem 1 not only uses the more intuitive concepts of join width (rather than expressibility in L^k), but also extend the characterization to non-Boolean queries.

Unfortunately, we cannot easily turn Theorem 1 into an optimization method. While determining if a given graph G has treewidth k can be done in linear time, this depends on k being fixed. Finding the treewidth is known to be NP-hard [5]. An alternative strategy to minimizing the width of intermediate results is given by the *bucket-elimination* approach for constraint-satisfaction problems [13], which are equivalent to Boolean project-join queries [26]. We now rephrase this approach and extend it to general project-join queries. Assume that we are given an order x_1, \dots, x_n of the attributes of a query Q . We start by creating n “buckets”, one for each variable x_i . For an atom $r_i(x_{i_1}, \dots, x_{i_k})$ of the query, we place the relation r_i with attributes x_{i_1}, \dots, x_{i_k} in bucket $\max\{i_1, \dots, i_k\}$. We now iterate on i from n to 1, eliminating one bucket at a time. In iteration i , we find in bucket i several relations, where x_i is an attribute in all these relations. We compute their join, and project out x_i if it is not in the target schema. Let the result be r'_i . If r'_i is empty, then the result of the query is empty. Otherwise, let j be the largest index smaller than i , such that x_j is an attribute of r'_i ; we move r'_i to bucket j . Once all the attributes that are not in the target schema have been projected out, we join the remaining relations to get the answer to the query. For Boolean queries (for which bucket elimination was originally formulated), the answer to the original query is ‘yes’ if none of the joins returns an empty result.

The maximal arity of the relations computed during the bucket-elimination process is called the *induced width* of the process relative to the given variable order. Note that the sequence of operations in the bucket-elimination process is independent of the actual relations and depends only on the relation’s schemas (i.e., the attributes) and the order of the variables [17]. By permuting the variables we can perhaps minimize the induced width. The *induced width of the query* is the induced width of bucket elimination with respect to the best possible variable order.

Theorem 2. *The induced width of a project-join query is its treewidth.*

This theorem extends the characterization in [15, 17] for Boolean project-join queries.

Theorem 2 tells us that we can optimize the width of intermediate results by scheduling the joins and projections according to the bucket-elimination process, using a subquery for each bucket, if we are provided with the optimal variable order. Unfortunately, since determining the treewidth is NP-hard [5], it follows from Theorem 2 that finding optimal variable order is also NP-hard. Nevertheless, we can still use the bucket-elimination approach, albeit with a heuristically chosen variable order. We follow here

the heuristic suggested in [7, 29, 30], and use the *maximum-cardinality search order* (MCS order) of [31]. We first construct the join graph G_Q of the query. We now number the variables from 1 to n , where the variables in the target schema are chosen as initial variables and then the i -th variable is selected to maximize the number of edges to variables already selected (breaking ties randomly).

6 Experimental Results

6.1 Implementation Issues

Given a k -COLOR graph instance, we convert the formula into an SQL query. For the non-Boolean case, before we convert the formula we pick 20% of the vertices randomly to be free. The query is then sent to the PostgreSQL backend, which returns a nonempty answer iff the graph is k -colorable. Most of the implementation issues arise in the construction of the optimized query from the graphs formulas. We describe these issues here.

The *naive* implementation of this conversion starts by creating an array E of edges, where for each edge $E[i]$ we store its vertices. We also construct an array min_occur such that for every vertex v_j , we have that $min_occur[j] = \min\{k | j \in E[k]\}$ is the minimal edge containing an occurrence of j . For the SQL query, we SELECT the first vertex that occurs in an edge, and list in the FROM section all the edges and their vertices using the *edge* relation. Finally, for the WHERE section we traverse the edges and for each edge E_i and each vertex $v_j \in E[i]$ we add the condition $E_i.v_j = E_l.v_j$, where $l = min_occur[j]$. For the non-Boolean case, the only change is to list the free vertices in the SELECT clause. The *straightforward* implementation uses the same data structures as the naive implementation. The SELECT statement remains the same, but the FROM statement now first lists the edges in reverse order, separated by the keyword JOIN. Then, traversing E as before, we list the equalities $E_i.v_j = E_l.v_j$ as ON conditions for the JOINS (parentheses are used to ensure the joins are performed in the same order as in the naive implementation).

For the *early-projection* method, we create another array max_occur such that for every vertex v_j , we have $max_occur[j] = \max\{k | j \in E[k]\}$ is the last edge containing v_j . Converting the formula to an SQL query starts the same way as in the straightforward implementation, listing all the edges in the FROM section in reverse order. Before listing an edge $E[i]$ we check whether $i \in max_occur[j]$ for some vertex v_j (we sort the vertices according to max_occur to facilitate this check). In such a case, we generate a subquery recursively. In the subquery, the SELECT statement lists all *live* vertices at this point minus the vertex v_j , effectively projecting out v_j . We define the set of live vertices as all the vertices $k \in E[l]$ such that $min_occur[j] \leq l \leq max_occur[j]$. This subquery is then given a temporary name and is joined with the edges that have already been listed in the FROM section. In the non-Boolean case we initially set, for a free vertex v_j , the $max_occur[j] = |E| + 1$. This effectively keeps these vertices *live*, so they are kept throughout the subqueries and they are then listed in the outer most SELECT clause. The *reordering* method is a variation of the early-projection method. We first construct a permutation π of the edges, using the greedy approach described in

Section 4. We then follow the early-projection method, processing the edges according to the permutation π .

Finally, for the *bucket-elimination* method, we first find the MCS order of the vertices (see Section 5). Given that order, for each vertex v_j we create a bucket that stores the edges and subqueries for v_j . The live vertices of the bucket are the vertices that appear in the edges and in the SELECT sections of the subqueries. Initially, the bucket for v_j contains all edges E_i such that v_j is the maximal vertex in E_i . We now create the SQL query by processing the buckets in descending order. A bucket for v_j is processed by creating a subquery in which we SELECT all the live variables in the bucket (minus v_j if v_j is not free), then listing all the edges and subqueries of the bucket in the FROM section of the subquery and JOINing them using ON conditions as in the straightforward approach. The subquery is then given a temporary name and moved to bucket $v_{j'}$, where $v_{j'}$ is the maximal vertex in the SELECT section of the subquery. The final query is obtained by processing the last bucket, SELECTing only one vertex in the Boolean case or SELECTing the free vertices in the non-Boolean case.

6.2 Density and Order Scaling

In order to test the scalability of our optimization techniques, we varied two parameters of the project-join queries we constructed. In these experiments we measured query execution time (recall that query compilation time is significant only for the straightforward approach). In the first experiment, we fixed the order of the 3-COLOR queries to 20 and compared how the optimizations performed as the density of the formula increased. This tested how, as the structure of the queries changes, the methods used scaled with the structural change. Figure 3 shows the median running times (logscale) of each optimization method as we increase the density of the generated 3-COLOR instances. The left side shows the Boolean query, and the right side considers the non-Boolean case with 20% of the vertices in the target schema. The shape of the curve for the greedy methods are roughly the same. At first, running time increases as density increases, since the number of joins increases with the density. Eventually, the size of the intermediate result becomes quite small (or even empty), so additional joins have little effect on overall running time.

Note that at low densities each optimization method improves upon the previous methods. The sparsity of the instances at low densities allow more aggressive application of the early projection optimization. For denser instances, these optimizations lose their effectiveness, since there are fewer opportunities for early projection. Nevertheless, bucket elimination was able to find opportunities for early projection even for dense instances. As the plot shows, bucket elimination completely dominates the greedy methods in running time for both the underconstrained and overconstrained regions. The non-Boolean instances demonstrate a similar behaviour.

For the next experiment, we fixed the density, and looked to see how the optimizations scaled as order is increased. For 3-COLOR we looked at two densities, 3.0 and 6.0; the lower density is associated with (most likely) 3-colorable queries while the high density is associated with (most likely) non-3-colorable queries. Figure 4 shows the running times (logscale) for several optimization methods for density 3.0 as order is scaled from 10 to 35. Notice that all the methods are exponential (linear slope in

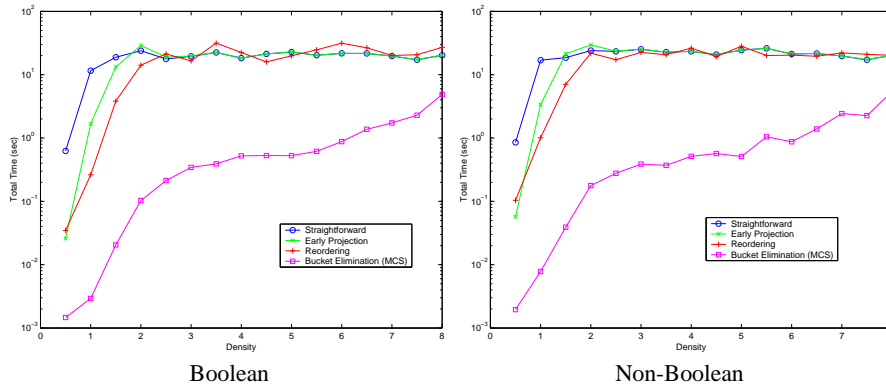


Fig. 3. 3-COLOR Density Scaling, Order = 20 – Logscale

logscale), but bucket elimination maintains a lower slope. This means that the exponent of the method is strictly smaller and we have an exponential improvement. Figure 5 shows the experiment for density 6.0 as order is scaled from 15 to 30. We see that the greedy heuristics do not improve upon the straightforward approach for both the underconstrained and the overconstrained densities, while bucket elimination still finds opportunities for early projection and shows exponential improvement over the other optimizations.

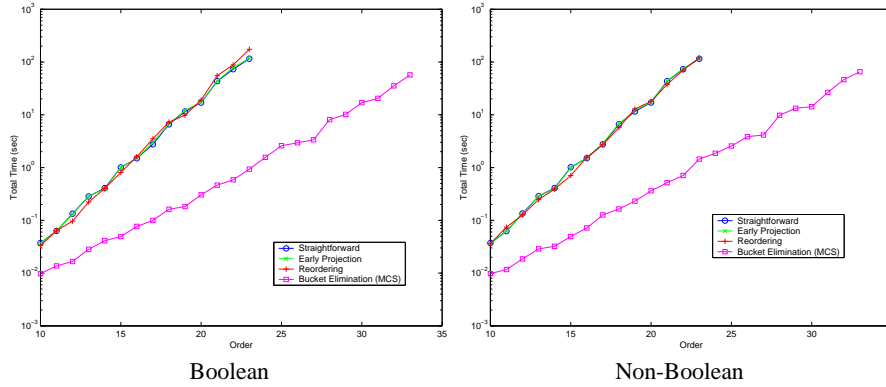


Fig. 4. 3-COLOR Order Scaling, Density = 3.0 – Logscale

Our next focus was to run order-scaling experiments for the structured queries. Figure 6 shows the running time (logscale) for the optimization methods as the augmented path instances scaled. The early projection and bucket elimination methods dominate. Unlike the random graph problems, early projection is competitive for these instances because the problem has a natural order that works well. But bucket elimination is still able to run slightly better. Here we start to see a considerable difference between the Boolean and non-Boolean case. The optimizations do not scale as well when we move to the non-Boolean queries. This is due to the fact that there are 20% less vertices to

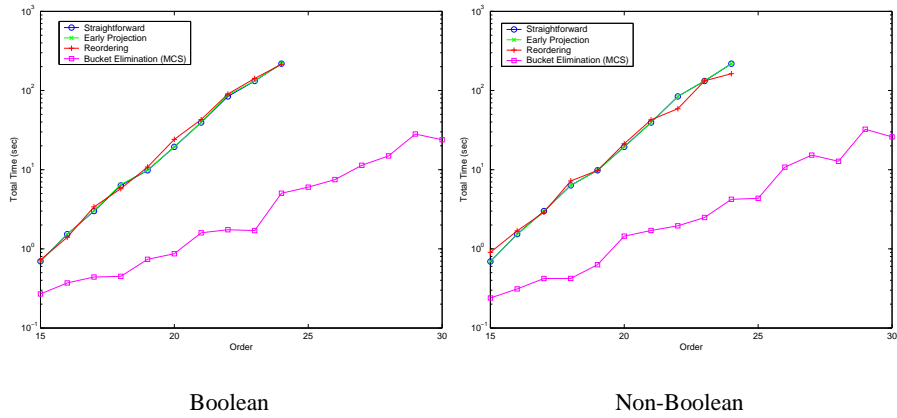


Fig. 5. 3-COLOR Order Scaling, Density = 6.0 – Logscale

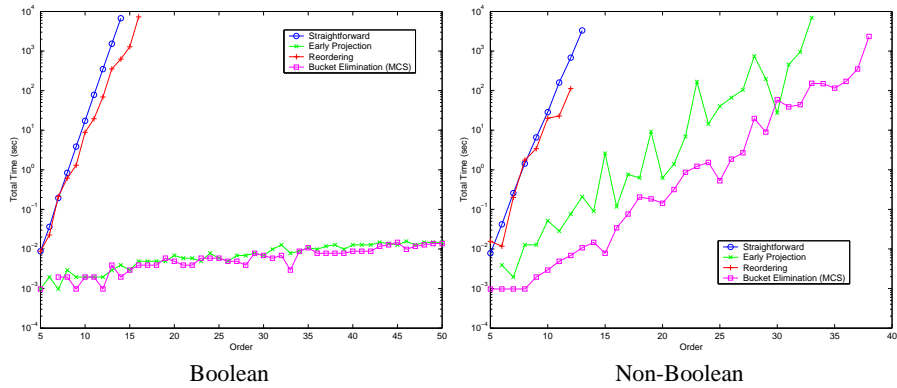


Fig. 6. 3-COLOR Augmented Path Queries – Logscale

exploit in the optimization, and each optimization method suffers accordingly. It should be noted that early projection and bucket elimination still dominate over the straightforward approach. Figure 7 shows the running time (logscale) for the methods when applied to the ladder graph instances. These results are very similar to the augmented path results, but notice that now reordering is even slower than the straightforward approach. At this point, not only is the heuristic unable to find a better order, but it actually finds a worse one. As we move to the augmented ladder instances in Figure 8 and the augmented circular ladder instances in Figure 9, the differences between the optimization methods become even more stark, with the straightforward and reordering methods timing out at around order 7. Also the difference between the Boolean and non-Boolean case becomes more drastic with the non-Boolean case struggling to reach order 20 with the faster optimization methods. But throughout both the random and structured graph instances, bucket elimination manages to dominate the field with an exponential improvement at every turn.

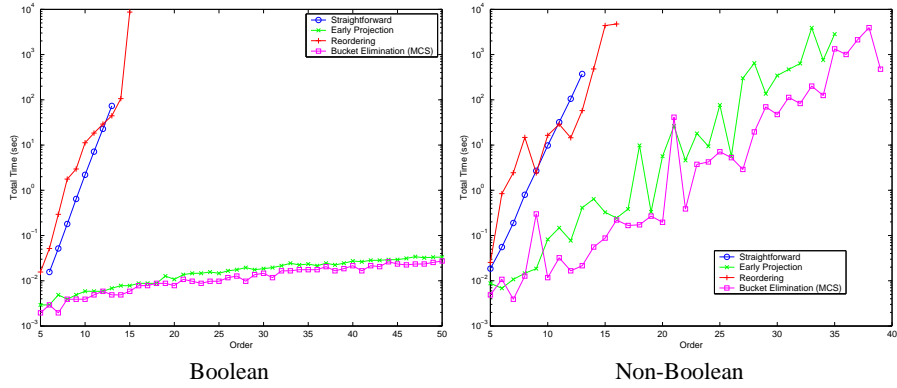


Fig. 7. 3-COLOR Ladder Queries – Logscale

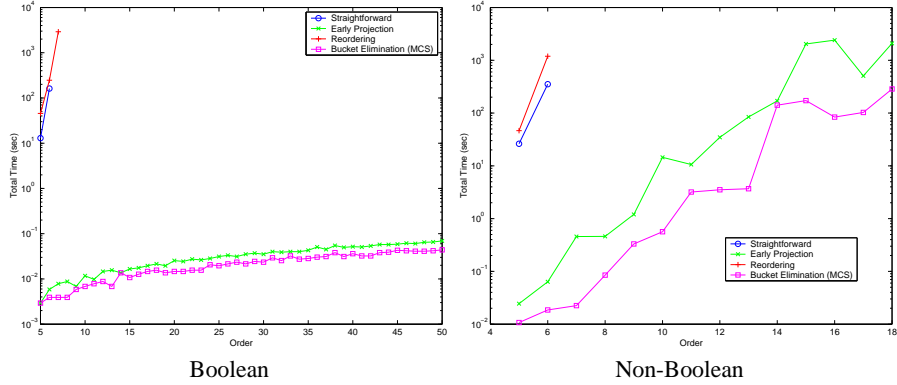


Fig. 8. 3-COLOR Augmented Ladder Queries – Logscale

7 Concluding Remarks

We demonstrated in this paper some progress in moving structural query optimization from theory, as developed in [10, 21, 26, 11], to practice, by using techniques from constraint satisfaction. In particular, we demonstrated experimentally that projection pushing can yield an exponential improvement in performance, and that bucket elimination yields an exponential improvement not only over naive and straightforward approaches, but also over various greedy approaches. In this paper, we focused on queries constructed from 3-COLOR; we have also tested our algorithms on queries constructed from 3-SAT and 2-SAT and have obtained results that are consistent with those reported here. This shows that NP-hardness results at the heart of structural query optimization theory need not be considered an insurmountable barrier to the applicability of these techniques.

Our work is, of course, merely the first step in moving structural query optimization from theory to practice. First, further experiments are needed to demonstrate the benefit of our approach for a wider variety of queries. For example, we need to consider relations of varying arity and sizes. In particular, one need is to study queries that could arise in mediator-based systems [36] as well as to study scalability with respect to rela-

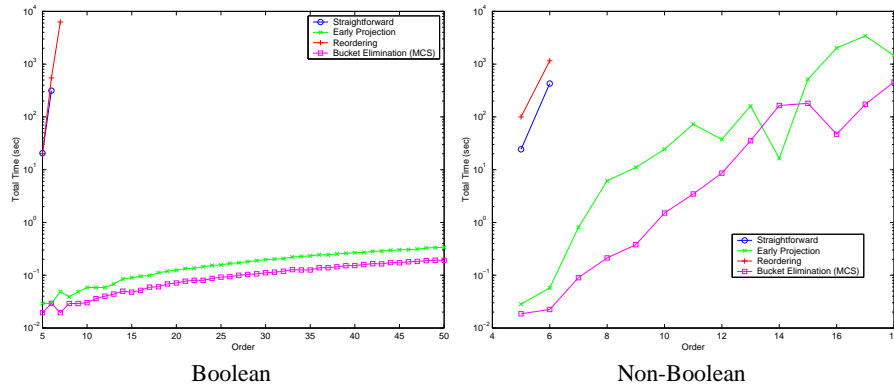


Fig. 9. 3-COLOR Augmented Circular Ladder Queries – Logscale

tion size. Second, further ideas should be explored, from the theory of structural query optimization, e.g., semijoins [34] and hypertree width [21], from constraint satisfaction, e.g., mini-buckets [12], from symbolic model checking, e.g., partitioning techniques [9], and from algorithmic graph theory, e.g., treewidth approximation [6]. Third, heuristic approaches to join *minimization* (as in [8]) should also be explored. Since join minimization requires evaluating a conjunctive query over a *canonical query database* [8], the techniques in this paper should be applicable to the minimization problem, cf. [27]. Fourth, structural query optimization needs to be combined with cost-based optimization, which is the prevalent approach to query optimization. In particular, we need to consider queries with *weighted* attributes, reflecting the fact that different attributes may have different widths in bytes. In particular, a variety of practical settings needs to be explored, such as nested queries, “exists” subqueries, and the like. Finally, there is the question of how our optimization technique can be integrating into the framework of rule-based optimization [18].

Acknowledgement: We are grateful to Phokion Kolaitis for useful discussions and comments.

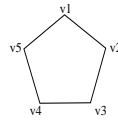
References

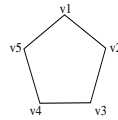
1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995.
2. A. Aho, Y. Sagiv, and J.D. Ullman. Efficient optimization of a class of relational expressions. *ACM Trans. on Database Systems*, 4:435–454, 1979.
3. A. Aho, Y. Sagiv, and J.D. Ullman. Equivalence of relational expressions. *SIAM Journal on Computing*, 8:218–246, 1979.
4. P. Apers, A. Hevner, and S. Yao. Optimization algorithms for distributed queries. *IEEE Trans. Software Engineering*, 9(1):57–68, 1983.
5. Arnborg, Corneil, and Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algebraic and Discrete Methods*, 8(2):277–284, 1987.
6. H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.
7. F. Bouquet. *Gestion de la dynamique et énumération d’implicants premiers: une approche fondée sur les Diagrammes de Décision Binaire*. PhD thesis, Université de Provence, France, 1999.

8. A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th ACM Symp. on Theory of Computing*, pages 77–90, 1977.
9. P. Chauhan, E.M. Clarke, S. Jha, J.H. Kukula, H. Veith, and Dong Wang. Using combinatorial optimization methods for quantification scheduling. In *Proc. 11th Conf. on Correct Hardware Design and Verification Methods*, pages 293–309, 2001.
10. C. Chekuri and A. Ramajaran. Conjunctive query containment revisited. Technical report, Stanford University, November 1998.
11. V. Dalmau, P.G. Kolaitis, and M.Y. Vardi. Constraint satisfaction, bounded treewidth, and finite-variable logics. In *Proc. Principles and Practice of Constraint Programming (CP'2002)*, pages 311–326, 2002.
12. R. Dechter. Mini-buckets: A general scheme for generating approximations in automated reasoning. In *International Joint Conference on Artificial Intelligence*, pages 1297–1303, 1997.
13. R. Dechter. Bucket elimination: a unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.
14. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
15. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.
16. R.G. Downey and M.R. Fellows. *Parametrized Complexity*. Springer-Verlag, 1999.
17. E.C. Freuder. Complexity of k -tree structured constraint satisfaction problems. In *Proc. AAAI-90*, pages 4–9, 1990.
18. J.C. Freytag. A rule-based view of query optimization. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 173–180, 1987.
19. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.
20. M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, 1979.
21. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *Proc. 18th ACM Symp. on Principles of Database Systems*, pages 21–32, 1999.
22. P.P. Griffiths, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.
23. A. Halevy. Answering queries using views: A survey. *VLDB Journal*, pages 270–294, 2001.
24. R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *Proc. 1996 Int'l Conf. on Computer Design*, pages 12–19, 1996.
25. Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *ACM SIGMOD International Conference on Management of Data*, pages 9–22, 1987.
26. Ph.G. Kolaitis and M.Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, pages 302–332, 2000. Earlier version in: Proc. 17th ACM Symp. on Principles of Database Systems (PODS '98).
27. I.K. Kunen and D. Suciu. A scalable algorithm for query minimization. Technical report, University of Washington, 2002.
28. R. Ramakrishnan, C. Beeri, and R. Krishnamurthi. Optimizing existential datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, 1988.
29. I. Rish and R. Dechter. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.
30. A. San Miguel Aguirre and M.Y. Vardi. Random 3-SAT and BDDs – the plot thickens further. In *Proc. Principles and Practice of Constraint Programming (CP'2001)*, pages 121–136, 2001.

31. R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to tests chordality of graphs, tests acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. on Computing*, 13(3):566–579, 1984.
32. J. D. Ullman. *Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, 1989.
33. M.Y. Vardi. On the complexity of bounded-variable queries. In *Proc. 14th ACM Symp. on Principles of Database Systems*, pages 266–276, 1995.
34. E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Trans. on Database Systems*, 1(3):223–241, 1976.
35. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7 Int'l Conf. on Very Large Data Bases*, pages 82–94, 1981.
36. R. Yerneni, C. Li, J.D. Ullman, and H. Garcia-Molina. Optimizing large join queries in mediation systems. *Lecture Notes in Computer Science*, 1540:348–364, 1999.

A Example Conversions



Consider the graph representing a pentagon: . It converts into the following conjunctive query:

$$\pi_{v_1} \text{edge}(v_1, v_2) \bowtie \text{edge}(v_1, v_5) \bowtie \text{edge}(v_4, v_5) \bowtie \text{edge}(v_3, v_4) \bowtie \text{edge}(v_2, v_3)$$

The following section shows what the SQL for the conjunctive query above would look like given the different methods of conversion.

A.1 Naive

```
SELECT <DISTINCT> e1.v1
FROM edge e1 (v1,v2), edge e2 (v1,v5), edge e3 (v4,v5), edge e4 (v3,v4), edge e5
(v2,v3)
WHERE e1.v1 = e2.v1
AND e2.v5 = e3.v5
AND e3.v4 = e4.v4
AND e1.v2 = e5.v2
AND e4.v3 = e5.v3;
```

A.2 Straightforward

```
SELECT <DISTINCT> e4.v3
FROM edge e5 (v2,v3) JOIN (
  edge e4 (v3,v4) JOIN (
    edge e3 (v4,v5) JOIN (
      edge e2 (v1,v5) JOIN edge e1 (v1,v2)
      ON ( e1.v1 = e2.v1 ))
    ON ( e2.v5 = e3.v5 ))
  ON ( e3.v4 = e4.v4 ))
ON ( e1.v2 = e5.v2 AND e4.v3 = e5.v3 );
```

A.3 Early Projection

```
SELECT <DISTINCT> e5.v3
FROM edge e5 (v2,v3) JOIN (
  SELECT <DISTINCT> e4.v4, t3.v2, e4.v3
  FROM edge e4 (v3,v4) JOIN (
    SELECT <DISTINCT> e3.v5, e3.v4, t4.v2
    FROM edge e3 (v4,v5) JOIN (
      SELECT <DISTINCT> e2.v1, e2.v5, e1.v2
      FROM edge e2 (v1,v5) JOIN edge e1 (v1,v2)
      ON ( e1.v1 = e2.v1 )) AS t4
    ON ( t4.v5 = e3.v5 )) AS t3
  ON ( t3.v4 = e4.v4 )) AS t2
ON ( t2.v2 = e5.v2 AND t2.v3 = e5.v3 );
```

A.4 Reordering

```
SELECT <DISTINCT> e1.v2
FROM edge e1 (v1,v2) JOIN (
  SELECT <DISTINCT> e3.v4, e3.v5, t3.v1, t3.v2
  FROM edge e3 (v4,v5) JOIN (
    SELECT <DISTINCT> e4.v3, e4.v4, e2.v5, e2.v1, e5.v2
    FROM edge e4 (v3,v4) JOIN (edge e5 (v2,v3) JOIN edge e2 (v1,v5)
    ON (TRUE )) ON ( e5.v3 = e4.v3 )) AS t3
  ON ( t3.v4 = e3.v4 AND t3.v5 = e3.v5 )) AS t2
ON ( t2.v1 = e1.v1 AND t2.v2 = e1.v2 );
```

A.5 Bucket Elimination

```
SELECT <DISTINCT> e3.v4
FROM edge e3 (v4, v5) JOIN (
  SELECT <DISTINCT> e4.v4, t1.v5
  FROM edge e4 (v3, v4) JOIN (
    SELECT <DISTINCT> e2.v5, t3.v3
    FROM edge e2 (v1, v5) JOIN (
      SELECT <DISTINCT> e1.v1, e5.v3
      FROM edge e1 (v1, v2) JOIN edge e5 (v2, v3)
      ON (e5.v2 = e1.v2)) AS t3
    ON (t3.v1 = e2.v1)) AS t1
  ON (t1.v3 = e4.v3)) AS t5
ON (t5.v4 = e3.v4 AND t5.v5 = e3.v5);
```