Structural Heuristics For Query Optimization

by

Benjamin James McMahan

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree
Master of Science

Approved, Thesis Committee:

Dr. Moshe Y. Vardi (Chair),
Karen Ostrum George Professor,
Computer Science

Dr. Devika S. Subramanian,
Professor,
Computer Science

Dr. Alan Cox,
Associate Professor,
Computer Science

Houston, Texas

March, 2004
Structural Heuristics For Query Optimization

Benjamin James McMahan

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. But as the number of joins increases, the size of the search space grows exponentially. Another approach to the problem, one that has been successful in constraint satisfaction, is that of structural optimization. The focus is on project-join orders that minimize the size of intermediate results. This thesis shows how structural techniques, including projection pushing and join reordering, can yield exponential improvements in query execution time. Finally, an implementation of the bucket elimination method is used to obtain another exponential improvement.
I would like to thank my advisor, Moshe Vardi. He has provided me with wisdom, insight, and, through his connections, friends around the world who also share my joy for logic. He has been able to push me when I turned idle and was very patient as I attempted to travel up the steep learning curve I needed to conquer to do the kind of research worthy of a degree.

I would also like to thank the other members of my committee. Every time I talk with Devika Subramanian she is able to capture my interest through her enthusiasm. I have also learned a great deal about how to tackle problems and how to make the most of the tools we have. I’ve always enjoyed talking with Alan Cox. When I was his teaching assistant I always learned something new whenever we spoke. I would also like to thank them for reading/scrutinizing over my thesis and provided me with advice on how to approach it.

A lot of what I do I owe to my parents. Extremely supportive, I always leave their presence motivated and energized. I will always want to be like my Dad and I will always enjoy talking with my Mom. Thanks for being always being there.

I need to thank Guoqiang Pan with whom I have had countless conversations, many of which pertained and was invaluable to my research. I would also like to thank David Schwarz and Konstantinos Bekris for putting up with my selections of movies and restaurants and for understanding when I wanted to pull my hair out. Finally, I would like to thank Priya Gupta, who has provided me with a reason to smile every day I come home from work.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi

1 Introduction 1
  1.1 Prior Work .................................................. 1
    1.1.1 Cost-Based Optimization ................................ 2
    1.1.2 Structural Methods ..................................... 4
  1.2 Contributions .................................................. 5
  1.3 Outline of the Thesis ........................................... 7

2 Experimental Test Suite 8
  2.1 Random k-SAT ................................................... 10
  2.2 Random k-COLOR ............................................... 12
  2.3 Structured Graphs ............................................... 13

3 Projection Pushing 15
  3.1 Naive Approach ............................................... 15
    3.1.1 Straightforward Approach ................................ 17
  3.2 Early Projection Approach ..................................... 18
  3.3 Reordering Approach ............................................ 20
  3.4 Bucket Elimination Approach .................................. 21
    3.4.1 Join-Expression Trees .................................... 22
    3.4.2 Bucket Elimination ......................................... 30
### 3.5 Experimental Results

- **3.5.1 Density Scaling**
- **3.5.2 Order Scaling**
- **3.5.3 Color Scaling**
- **3.5.4 Structured Queries**

### 4 Tree Decomposition Heuristics

- **4.1 Induced Width = Tree Width**
- **4.2 Variable Order Heuristics**
  - **4.2.1 Maximum Cardinality Search**
  - **4.2.2 Lexicographic Breadth First Search**
- **4.3 Experimental Results**
  - **4.3.1 Density Scaling**
  - **4.3.2 Order Scaling**
  - **4.3.3 Structured Queries**

### 5 Conclusions
Illustrations

2.1 Augmented path, ladder, augmented ladder, and augmented circular ladder ............................................. 14

3.1 Naive and straightforward approaches: density scaling of compile time, 5 variables, logscale ................................................ 17

3.2 3-COLOR Density Scaling, Order = 20 – Logscale ....................... 33

3.3 3-SAT Density Scaling, Order = 20 ........................................ 34

3.4 3-COLOR Order Scaling, Density = 3.0 – Logscale ...................... 35

3.5 3-COLOR Order Scaling, Density = 6.0 – Logscale ...................... 36

3.6 3-SAT Order Scaling, Density = 2.0 – Logscale ......................... 36

3.7 3-SAT Order Scaling, Density = 6.0 – Logscale ......................... 37

3.8 n-COLOR – Order = 5, Density = 1.0 ...................................... 38

3.9 3-COLOR Augmented Path Queries – Logscale .......................... 38

3.10 3-COLOR Ladder Queries – Logscale .................................... 39

3.11 3-COLOR Augmented Ladder Queries – Logscale ..................... 39

3.12 3-COLOR Augmented Circular Ladder Queries – Logscale ........... 40

4.1 3-COLOR Density Scaling, Order = 20 – Logscale ...................... 51

4.2 3-SAT Density Scaling, Order = 20 – Normal Scale .................... 51

4.3 3-COLOR Order Scaling, Density = 3.0 – Logscale .................... 52

4.4 3-COLOR Order Scaling, Density = 6.0 – Logscale .................... 53

4.5 3-SAT Order Scaling, Density = 2.0 – Logscale ....................... 53
4.6  3-SAT Order Scaling, Density = 6.0 – Logscale  . . . . . . . . . . . . .  54
4.7  3-COLOR Augmented Path Queries – Logscale  . . . . . . . . . . . . .  54
4.8  3-COLOR Ladder Queries – Logscale  . . . . . . . . . . . . . . . . . .  55
4.9  3-COLOR Augmented Ladder Queries – Logscale  . . . . . . . . . . . .  55
4.10 3-COLOR Augmented Circular Ladder Queries – Logscale . . . . . . .  56
Chapter 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 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 [Ull89, GMUW00]. For example, consider a 10-way join between the relations $r_1, \ldots, r_{10}$. Let $r_1, \ldots, r_9$ contain 50 attributes, millions of tuples, and a high amount of similarity in the attributes joined between them. Finally let $r_{10}$ be an empty relation. It is easy to see that the order in which we choose to join the tables is extremely important as the 10-way join could practically be a constant time operation if $r_{10}$ is contained in the first join since its emptiness will propagate through the remaining joins. But if $r_{10}$ is the last relation to be joined, the database will have filled up gigabytes worth of data only to find out it must delete it all and return nothing. There are three methods of database optimizations found in the literature: cost-based, join minimization, and structural optimization. This thesis focuses structural optimization, bringing in methods from Artificial Intelligence, and applying them in a database setting with good empirical results.

1.1 Prior Work

The standard approach to query optimization is that of cost-based optimization [GAC+79]. This approach requires the development of a cost model 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. Cost-based optimizations are effective when the queries, and the corresponding search space of plans, are small, but they do not scale up well as the query grows in size.

Two other approaches found in the database-theory literature had little impact on query-optimization practice. The first approach, initially proposed by Chandra and Merlin in [CM77] and then explored further by Aho, Sagiv, and Ullman in [ASU79b, ASU79a], 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 [GJ79], and has not been pursued in practical query processing.

The second optimization approach, the one this thesis focuses on, uses the structural properties of a 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 [WY76], was first analytically studied for acyclic joins [Yan81], 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. This thesis extends the work on structural query optimization to close the gap between its theoretical potential and its actual usage in a database optimizer.

1.1.1 Cost-Based Optimization

The standard approach to selecting a project-select-join order is that of cost-based optimization [GAC+79]. The major paradigm is to consider the choice of an efficient plan as a search problem. This search problem requires that the optimizer have

- a set of semantically correct plans called the search space,

- a cost estimation function that can assign a cost for evaluation to each plan in the search space, and

- an enumeration algorithm that can list plans in the search space.
The goal of any optimizer is to be able to find the low costing plans in the search space using an accurate cost estimation function and a fast enumeration algorithm [Cha98].

The cost estimation function requires the development of a cost model. This cost model uses database statistics such as relation size, block size, and selectivity, to assign a cost to basic blocks (such as the relations) of the query. The model is then extended to include formulas which estimate of the cost of operations applied to the relations. For example, the size of the output of a join could be estimated as the product of the size of the two relations and then trimmed using the selectivity knowledge the optimizer has about the attributes that were joined on. Typically, the costs correspond to I/O costs, in particular disk time, but could also include CPU costs, memory/cache latency, network latency, and any other resource bottlenecks the optimizer forsees as a problem.

Once a cost estimation function is developed, 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. [AHY83]), or incomplete, such as simulated annealing, genetic algorithms, or other approaches (cf. [IW87]). 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. [WY76]). In particular, selections and projections can be “pushed” downward, reducing the number of tuples and columns in intermediate relations [Ull89]. A naive approach at enumerating such plans can lead to $O(n!)$ plans. For example, [LOT94] draws an upper bound on the number of distinct join orders for $n$ joins as

$$\binom{2(n-1)}{n-1} \times \frac{(n-1)!}{2^{n-1}}.$$

As a rule, optimizers make use of dynamic programming to reduced the number of plans to enumerate. This paradigm requires an assumption that for $n$ joins, it is sufficient to consider all subqueries with $n-1$ joins and to extend those plans with
another join. The dynamic programming then proceeds bottom-up, by considering at each $i$th step all the optimal plans for subqueries of $i$ joins. This approach allows us to eliminate the factorial in our upper bound of possible plans and leaves us with only $O(n2^{n-1})$ plans to consider [Cha98]. In general, 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. Due to this fact, we propose structural optimizations that do not have this problem of scalability.

1.1.2 Structural Methods

The focus of this thesis is the use of structural methods to find an optimal execution plan for a query. This method first appeared in [WY76], which analytically studied the use of structural properties for acyclic joins [Yan81]. 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 [Yan81], the approach focuses on choosing the project-join order in such a manner so as to polynomially bound the size of intermediate results [CR98, GLS99, KV00, DKV02]. For example, [DKV02] 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 \textit{join graph}, which consists of all attributes as nodes and all relation schemes in the join as cliques. It is shown in [DKV02] that the minimal arity is the \textit{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 [DF99] (for a formal definition see Chapter 3.4.1). The arity of intermediate relations is a good proxy for their size, since a constant-arity bound translates to a polynomial-size bound. This bound reveals a theoretical limit on the effectiveness of projection pushing and join reordering in terms of the treewidth of the join graph. While acyclicity can be tested efficiently [TY84], finding the treewidth of a graph is NP-hard [ACP87]. Thus, the results in [DKV02] 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 [CR98, GLS99, KV00, DKV02], has not contributed to query optimization in practice.

However, the projection-pushing strategy has been applied to solve constraint-satisfaction problems in Artificial Intelligence with good experimental results [RD00, SV01]. 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 [Dec03]. A general method for projection pushing in the context of constraint satisfaction is the bucket-elimination method [DP87, Dec03]. Since evaluating Boolean project-join queries is essentially the same as solving constraint-satisfaction problems [KV00], this thesis studies the application of 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.

1.2 Contributions

In order to focus solely on projection pushing in project-join expressions, we choose an experimental setup in which the cost-based approach is 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 [YLUGM99]. They challenge cost-based planners because of the exceedingly large size of the search space, which leads 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 irrel-
evant. Such databases arise naturally in query containment and join minimization, where the query itself is viewed as a database [CM77]. For a survey of recent applications of query containment see [Hal01].

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 [KV00] to generate queries and databases corresponding to instances of 3-COLOR and 3-SAT problems [GJ79]. Our main focus in this thesis is to compare 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. In addition to random 3-COLOR and 3-SAT problems, we use a variety of queries with specific structures, such as “augmented paths,” “ladders,” “augmented ladders,” and “augmented circular ladders” [KS02].

We compare the scalability of the bucket-elimination method with the naive, straightforward, early projection, and reordering method. The naive and straightforward plan join the relations in the order in which they are listed, without applying projection pushing. Early projection then proceeds to apply rojection pushing and reordering applies a new join order, all done 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 re-ordering 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 [DP87, Fre90, KV00, DKV02]). Since
finding such an order is NP-hard [ACP87], we use the “maximum cardinality” order of [TY84], which is often used to approximate the optimal order [Bou99, RD00, SV01]. 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 that it significantly dominates greedy heuristics, without incurring the excessive cost of searching large plan spaces.

The highly effective bucket elimination results were coupled with maximum cardinality order, therefore we consider new variable orders to compare if the gain in scalability was from the bucket elimination method, the maximum cardinality order, or from both. To aid us in finding new heuristics, we first provide algorithms to convert from tree decompositions to variable orders that provide equivalent widths. We then consider two variants of a lexicographic breadth first search heuristics used in finding tree decompositions. We demonstrate experimentally that the maximum cardinality order still provides the best approach, but that the lexicographic orders are more competitive than the greedy heuristics used earlier.

1.3 Outline of the Thesis

The outline of the thesis is as follows. Chapter 2 describes the experimental setup. We then describe a naive and straightforward approach in Chapter 3, a greedy heuristic approach in Chapter 3, and the bucket elimination approach in Chapter 3. We report on our scalability experiments for these optimizations in Chapter 3. Next, Chapter 4 provides algorithms for converting between tree decompositions and variable orders for the bucket elimination methods. Using these algorithms Chapter 4 considers more heuristics to use with the bucket elimination method with experimental results in Chapter 4. Finally, we conclude with a discussion in Chapter 5.
Chapter 2

Experimental Test Suite

The focus of this thesis is to use the structure of the query to drive the query optimization. In order to focus on only structural issues of the optimization process, the experimental setting is one in which the traditional cost-based approach is ineffective. Thus, we focus on project-join queries with a large number of relations (up to and over 100). Such expressions are common in mediator-based systems [YLUGM99]. 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 index or selectivity information is rather useless. Hash joins proved to be the most effective, so to further reduce the optimizer’s choices we indicate to the database to only use hash joins on our queries. 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). Later, to simulate a typical database’s usage more closely, we also consider non-Boolean queries where a fraction (twenty percent) of the attributes are not projected out.

Our queries also have the following features:

- Projecting out a column from our relations yields a relation with all possible tuples. Thus, in our setting, semijoins, as in the Wong-Youssefi algorithm [WY76], are useless. Therefore, the thesis focuses solely on ordering joins and projections.

- Since all relations are of fixed arities, differences between the various notions of width, such as treewidth, query width, and hypertree widths are minimized
[CR98, GLS99], enabling us to focus on treewidth.

The thesis focuses on project-join queries, also known as conjunctive queries. 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 $\pi_{x_1,\ldots,x_n}(R_1 \bowtie \ldots \bowtie R_m)$, where the $R_j$s are relations and $x_1,\ldots,x_n$ are the free variables (we use the terms “variables” and “attributes” interchangably). Conjunctive queries are usually defined as positive, existential conjunctive first-order formulas [AHV95], but we focus of the relation algebra formulation for a more intuitive translation to later SQL queries.

Our experimental setup works as follows: we generate a conjunctive query, translate it into SQL according to a given optimization method, then run the query on the database collecting both the PLANNER time and EXECUTION time. We generate the conjunctive queries through both random and structured methods. The random methods correspond to random satisfiability and graph colorable instances. These provide various methods to see how our optimizations scale when the difficulty of the problem scales up. In particular, they provide a way to test for not only the scalability of the optimizations as the size of the query increases, but also as the structure of the query changes from under constrained to over constrained. Next, structured queries were generated corresponding to graph colorable instances. This allowed us to fix the structure of the query more absolutely, and test the scalability of the methods as the query grows in size. Structured queries also more closely correlate to the type of queries a database typically expects to see. It is important to note, however, that our algorithms do not rely on the special structure of the queries we generate – they are applicable to project-join queries in general.

All experiments were performed on the Rice Tersascale Cluster\textsuperscript{1}, which is a Linux cluster of Itanium II processors with 4GB of memory each, using PostgreSQL\textsuperscript{2} 7.2.1. For each run we measured the time it took to generate the query, the time the

\footnote{1\url{http://www.citi.rice.edu/rtc/}}\footnote{2\url{http://www.postgresql.org/}}
PostgreSQL Planner took to optimize the query, and the execution time needed to actually run the query. Due to the presence of heavy outliers in random satisfiability and graph coloring we report only median running times.

2.1 Random k-SAT

The first of the random queries we generated was random k-SAT. An instance of k-SAT $\phi$ is a formula in clausal normal form (CNF). That is, $\phi$ is a conjunction $\bigwedge_{i=1}^{m} c_i$ of clauses, where each clause $c_i$ is a disjunction $\bigvee_{l_{i,j}}$ of at most $k$ literals. A literal is either a Boolean variable or a negated Boolean variable. The k-SAT problem asks whether or not there is an assignment to the variables in $\phi$ such that $\phi$ becomes true, i.e., is $\phi$ satisfiable. The thesis focuses on results from both 3-SAT and 2-SAT.

To solve a k-SAT formula using a database, the database requires a set of relations with which to process the query generated from the formula. Without loss of generality we can assume that in each clause of a k-SAT formula the negative literals precede positive literals. Consider 3-SAT. There are four types of clauses, containing 0, 1, 2, or 3 negative literals. For example, $\neg p_3 \lor p_2 \lor p_7$ contains one negative literal. For a clause with $i$ negative literals, we define a ternary relation $r_i$ that contains all seven truth assignments that satisfy such a clause. For example, $r_1$ contains all 3-bit tuples except for $(1, 0, 0)$. We can now express satisfiability of $\phi$ by a conjunctive query $Q_\phi$ over the database consisting of the relations $\{r_0, \ldots, r_3\}$. Let the set of Boolean variables in $\phi$ be $\{p_1, \ldots, p_n\}$. The 3-SAT problem can then be expressed as the project-join query:

$$Q_\phi = \pi_0 \bowtie_{j=0}^{m} r_{i_j}(x_{s_j}, s_{t_j}, x_{u_j})$$

, where clause $c_j = l_{j,1} \lor l_{j,2} \lor l_{j,3}$ is of type $i_j$ and the variables $x_{s_j}, x_{t_j}, x_{u_j}$ corresponds to the Boolean variables of $l_{j,1} \lor l_{j,2} \lor l_{j,3}$. Each atom of the query corresponds to a clause in the Boolean formula. For example, the clause $\neg p_3 \lor p_2 \lor p_7$ of type 1 yields the atom $r_1(x_3, x_2, x_7)$. $Q_\phi$ returns a nonempty tuple if and only if $\phi$ is satisfiable. We also generated conjunctive queries from 2-SAT instances. In this case, we need
three binary relations, each consisting of three tuples, corresponding to clauses with 0, 1, or 2 negative literals.

Once the database is setup we can then generate k-SAT formulas to test the optimization methods on. In particular we generate random 3-SAT instances with varying structural properties. For a fixed number \( n \) of Boolean variables and a fixed number \( m \) of clauses, instances are generated uniformly. A clause is generated by choosing uniformly at random three distinct variables and then choosing their polarity uniformly at random. Clauses are generated (with repetitions) until the right number of clauses is arrived at. This is a standard distribution, called the \textit{fixed-width distribution}, in the study of random 3-SAT, cf. [SML96]. 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 \textit{order} (i.e., the number of variables) fixed, while scaling up the \textit{density}, which is the ratio \( m/n \) of clauses to variables. 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 satisfiable, while a high density suggests that the instance is overconstrained and is unlikely to be satisfiable. Experimental research has shown that for ratios below (roughly) 4.26, the probability of satisfiability of a random 3-SAT instance goes to 1 as the order increases, while for ratios above 4.26 the probability goes to 0; at 4.26, the probability of satisfiability is near 0.5 [SML96, CA96]. There is extensive theoretical research attempting to prove a sharp phase transition at density 4.26. Thus, density scaling yields a spectrum of problems, going from underconstrained to overconstrained.

To capture the difficulty of determining satisfiability in the context of random 3-SAT, both order and density scalability have been investigated. Density scaling has been studied extensively, cf. [SML96, CA96], which measured the effect of density scaling on the running time of algorithms based on the so-called \textit{Davis-Putnam-Logemann-Loveland method} (abbr., DPLL method), a depth-first search with unit propagation [DLL62]. Intuitively, underconstrained instances are easy to solve, as
a satisfying assignment can be found fast, and overconstrained instances are also less hard to solve, as all branches of the search terminate quickly. Indeed, the data displayed in [CA96, SML96] demonstrate a peak in running time essentially at the crossover density of 4.26. Order scaling has been studied in [CDS+00, CM01, SV01] for both DPLL-based algorithms, as well as for algorithms based on Boolean Decision Diagrams [Bry86]. The pattern that emerges there is that of
(1) polynomial-time scalability for low-density instances,
(2) exponential-time scalability for high-density instances, and
(3) algorithm-dependent transition density from polynomial to exponential scalability.

The behavior of random 2-SAT according to the fixed-width distribution is quite well understood. First, it is well-known that unlike 3-SAT, which is NP-complete, 2-SAT can be solve in polynomial time [Coo71]. Also, 2-SAT has a provable phase transition: the asymptotic probability is 1 for densities below 1 and 0 for densities above 1 [CR92]. Even the precise character of the phase transition is known [BCP01].

For 3-SAT, we study orders between 5 and 30 and integral densities between 1 and 9. For 2-SAT we study orders between 5 and 60 and densities between 0.2 and 5. For each point in the $d \times n$ plane that we consider, we generate 100 random formulas. We then report median running times (since the mean tends to be skewed by outliers).

2.2 Random k-COLOR

Another type of random query we generate is of k-COLOR graph instances. An instance of k-COLOR is a graph $G = (V, E)$ and a set of colors $C = \{1, 2, \ldots, k\}$, where $|V| = n$ and $|E| = m$. The problem k-COLOR asks whether or not there is a way to color $V$ using $C$ where for every $(u, v) \in E$, $c(u) \neq c(v)$. For each edge $(u, v) \in E$ there are $\binom{k}{2}$ ways of coloring $(u, v)$ to satisfy the requirement of no monochromatic edges. We define the relation $edge_k$ as containing tuples corresponding to all the pairs of distinct colors. For example, $edge_3 = \{< 1, 2 >, < 2, 1 >, < 1, 3 >, < 3, 1 >, < 2, 3 >, < 3, 2 >\}$ corresponding to the six possible colorings of the edges that maintain
the 3-COLOR property. The k-COLOR problem can then be expressed as the project-join query $Q_G = \pi_{\emptyset} \triangledown_{(v_i, v_j) \in E} edge_k(v_i, v_j)$. This query returns a nonempty result over the $edge_k$ relation iff $G$ is k-colorable [CM77].

To generate a range of queries with varying structural properties, we start by generating random graph instances using the random graph process $G(n, m)$. 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. As in the SAT problem, 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 k-colorable, while a high density suggests that the instance is overconstrained and is unlikely to be k-colorable. In particular, a random graph instance of density less than 4 is almost surely 3-colorable and an instance of density greater than 5 is almost surely non-3-colorable [Dem03]. Thus, density scaling yields a spectrum of problems, going from underconstrained to overconstrained. For 3-COLOR we generated orders between 10 and 35 and densities between 0.5 and 8.0, and for 2-COLOR we generated orders between 10 and 100 and densities between 0.5 and 8.0. As in the k-SAT case, for each point of interest in the $d \times n$ plane, we generate 100 random graphs and report the median running times.

2.3 Structured Graphs

Database queries usually are not randomly generated, and in turn we also consider non-random graph instances for 3-COLOR. These queries, suggested in [KS02], have specific structures. An augmented path (Figure 2.1a) is a path of length $n$, where for each vertex on the path a dangling edge extends out of the vertex. A ladder
(Figure 2.1b) is a ladder with \( n \) rungs. An \textit{augmented ladder} (Figure 2.1c) is a ladder where every vertex has an additional dangling edge, and an \textit{augmented circular ladder} (Figure 2.1d) 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.

Figure 2.1 : Augmented path, ladder, augmented ladder, and augmented circular ladder

These queries have fixed structural bounds that they keep as the size of the query grows. The augmented path is a tree and thus has a treewidth of one. The ladder corresponds to a treewidth of two. The augmented ladder also corresponds to a treewidth of two, but its structure is such that a heuristic might overestimate the dangling edges from the ladder. Finally, the augmented circular ladder has a treewidth of four. Thus, even when the size of the query is scaled, the treewidth of the query is fixed. The next chapter describes treewidth as a way to characterize a limit for structural optimizations, and this bound on the treewidth allows us to see if the optimizations are able to closely follow this limit.
Chapter 3

Projection Pushing

This chapter describes several optimizations for project-join orders of conjunctive queries using structural techniques. We start by considering naive conjunctive queries as a base case. Difficulties with compilation time lead us to consider straightforward conjunctive queries, which obtain the same execution time as the naive ones, but without the extraordinary compilation time. We then develop two greedy methods, early projection and reordering which focus on improving performance by eliminating unnecessary attributes whenever possible.

Next, the chapter changes focus slightly and looks at the theoretical limit of early projection through a concept of join-expression trees. A characterization of these join-expression trees is made. That characterization leads into the motivation for the use of the bucket elimination method from constraint satisfaction as an optimization method for project-join orders. This work is the first appearance of bucket elimination applied to database optimization, and we also extend the bucket elimination method to handle non-Boolean queries. The chapter then finished with experimental results used to compare the different execution times of the straightforward, early projection, reordering, and bucket elimination methods.

3.1 Naive Approach

Given a conjunctive query $\psi = \pi_0 \times_{i=0}^{m} r(v_{i1}, \ldots, v_{ik})$, we first use a naive translation of $\psi$ into SQL:
SELECT 1
WHERE EXISTS(
SELECT *
FROM $r_1 (v_1, \ldots, v_{k_1}), \ldots, r_m (v_{m_1}, \ldots, v_{m_k})$
WHERE $\bigwedge_{j=1}^m (r_{j}.v_j = r_{p(j)}(j).v_{j})$ AND \ldots AND $r_{j}.v_{j_k} = r_{p(j_k)}(j_k).v_{j_k}$;

As SQL does not explicitly allow us to express Boolean queries, we instead select 1 when the subquery corresponding to the conjunctive query is non-empty. The FROM section simply enumerates all the atoms in the query, referring to them as $r_1, \ldots, r_m$ and renames the columns to match the attributes of the query. The WHERE section enforces equality of different occurrences of the same attribute. More precisely, we enforce equality of each occurrence to the first occurrence of the same attribute; $p(v_i)$ points to the first occurrence of the attribute $v_i$.

For example, consider the conjunctive query for the following 3-COLOR pentagon graph instance: $\pi_0 edge (v_1, v_2) \bowtie edge (v_1, v_5) \bowtie edge (v_4, v_5) \bowtie edge (v_3, v_4) \bowtie edge (v_2, v_3)$. The resulting SQL query runs as follows:

SELECT 1
WHERE EXISTS (SELECT *
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);

We initially ran these queries for 3-SAT 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 3.1.
We used the PostgreSQL Planner’s genetic algorithm option to search for a query plan, because the exhaustive search for our queries turn out to be infeasible. The genetic algorithm still 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.

![Graph](image)

Figure 3.1 : Naive and straightforward approaches: density scaling of compile time, 5 variables, logscale

### 3.1.1 Straightforward Approach

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

```
SELECT 1
WHERE EXISTS ( SELECT *
FROM r r_1 (v_1, ..., v_{1k}) NATURAL JOIN ... NATURAL JOIN r_{m} (v_{m1}, ..., v_{mk}) )
```

Parentheses forces the evaluation to proceed from \( r_1 \) to \( r_2 \) and onwards (i.e., \((... (r_1 \Join r_2) ... \Join r_m)\)). We omit parentheses here for sake of readability.
As an example, consider the above 3-COLOR conjunctive query given by the pentagon graph instance. The conversion now appears as

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

The order in which the relations are listed then becomes the order in which 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 3.1, compile time still scales exponentially with density, but more gracefully than the compile time for the naive approach.

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.

The rest of the thesis focuses on 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.

### 3.2 Early Projection Approach

The conjunctive queries we are considering have the form $\pi_{v_1,\ldots,v_k}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_m)$. If $v_j \not\in \{v_1,\ldots,v_k\}$ does not appear in the relations $r_{q+1},\ldots,r_m$, then we can rewrite the formula into an equivalent one:
\[
\pi_{v_1,\ldots,v_k}(\pi_{\text{livevars}}(r_1 \bowtie \ldots \bowtie r_q) \bowtie \ldots \bowtie r_m),
\]
where \text{livevars} are all the attributes in the scope minus \(v_j\). This means we can write a query to join the relations \(r_1,\ldots,r_q\), project out \(v_j\), and join the result with relations \(r_{q+1},\ldots,r_m\). We then say that the projection of \(v_j\) has been pushed in and \(v_j\) has been projected early. The goal 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 projection subformula becomes a subquery. Suppose that \(q\) and \(j\) above are minimal. Then the SQL query can be rewritten as:

```sql
SELECT 1
WHERE EXISTS ( SELECT *
FROM r r_{m} (v_{m1},\ldots,v_{mk}) NATURAL JOIN \ldots NATURAL JOIN r r_{q+1} (v_{q+11},\ldots,v_{q+1k}) NATURAL JOIN ( subquery_q ) AS t_q
WHERE subquery_{k} is obtained by translating the subformula \(\pi_{\text{livevars}}(r_1 \bowtie \ldots \bowtie r_k)\) into SQL according to the straightforward approach. The only difference between the subquery and the full query is the SELECT section. In the subquery the SELECT section contains all \text{live} attributes within its scope, i.e., all attributes except for \(v_j\). Finally, we proceed recursively to apply early projection to the join of \(r_{q+1},\ldots,r_m\).

Consider the pentagon example \(\pi_\emptyset \text{edge}(v_1,v_2) \bowtie \text{edge}(v_1,v_5) \bowtie \text{edge}(v_4,v_5) \bowtie \text{edge}(v_3,v_4) \bowtie \text{edge}(v_2,v_3)\). We can rewrite the conjunctive query into the following early projection SQL query:
SELECT 1
WHERE EXISTS ( 
SELECT *
FROM edge e5 (v2,v3) NATURAL JOIN ( 
  SELECT <DISTINCT> e4.v4, t3.v2, e4.v3 
  FROM edge e4 (v3,v4) NATURAL JOIN ( 
    SELECT <DISTINCT> e3.v5, e3.v4, t4.v2 
    FROM edge e3 (v4,v5) NATURAL JOIN ( 
      SELECT <DISTINCT> e2.v1, e2.v5, e1.v2 
      FROM edge e2 (v1,v5) NATURAL JOIN edge e1 (v1,v2) 
    ) AS t4 
  ) AS t3 
) AS t2);

Note that the keyword DISTINCT can be added to help reduce the size of inter-
mediate results. The optimizer would have to weigh the benefits of obtaining smaller 
relations with the cost of sorting the relations to perform the DISTINCT operation. 
In general, we added DISTINCT to our queries as it proved to be the faster option.

3.3 Reordering Approach

The early projection method processes the relations of the query in a linear fashion. 
Since the goal of early projection is to project attributes as soon as possible, reordering 
the relations may enable us to project early more aggressively. For example, if an 
attribute $v_j$ appears only in the first and the last relation, early projection will not 
project $v_j$ 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 $r_1,\ldots,r_m$, we can apply a permutation $\rho$ and process the relations in 
the order $r_{\rho(1)},\ldots,r_{\rho(m)}$. The permutation $\rho$ should be chosen so as to minimize the 
number of live attributes in the intermediate relations. This observation was first 
made in the context of symbolic model checking (see [HKB96]). 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 attributes that occur only once in the remaining atoms. If there is a tie, the algorithm chooses the atom that shares the least attributes with the remaining atoms. Further ties are broken randomly.

Once the permutation $\rho$ is computed, we construct the same SQL query as before, but this time with the order suggested by $\rho$. We call this method reordering.

Using the pentagon example given above, the converted reordering SQL query appears as follows:

\[
\begin{align*}
\text{SELECT } 1 \\
\text{WHERE EXISTS (}
\text{SELECT } * \\
\text{FROM edge e1 (v1,v2) NATURAL JOIN (}
\text{SELECT <DISTINCT> e3.v4, e3.v5, t3.v1, t3.v2 \\
\text{FROM edge e3 (v4,v5) NATURAL JOIN (}
\text{SELECT <DISTINCT> e4.v3, e4.v4, e2.v5, e2.v1, e5.v2 \\
\text{FROM edge e4 (v3,v4) NATURAL JOIN (edge e5 (v2,v3) NATURAL JOIN edge e2 (v1,v5)))}
\text{) AS t3}
\text{) AS t2);}
\end{align*}
\]

### 3.4 Bucket Elimination Approach

The optimizations applied in Section 3.2 correspond to a particular rewriting of the original conjunctive query according to the algebraic laws of the relational algebra [Ull89]. 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?

### 3.4.1 Join-Expression Trees

Using join-expression trees, we show what the limit of projection pushing is in terms of a
notion called join width. We then equate (plus one) join width with the graph-
theoretic notion of treewidth. To do this we first define both join-expression trees
and tree decomposition. We then characterize join width in terms of treewidth and
prove the relationship between the two.

**Definition 3.1.** Consider a project-join query $Q = \pi_{x_1, \ldots, x_n}(R_1 \bowtie \ldots \bowtie R_m)$ 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 $S_Q = \{x_1, \ldots, x_n\}$ 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 $L_w : V_Q \to 2^A$ and $L_p : V_Q \to 2^A$
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 charac-
terization 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* [DF99].

**Definition 3.2.** 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 $k$ [DF99].

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

**Theorem 3.1.** The join width of a project-join query $Q$ is $tw(G_Q) + 1$.

**Proof:** 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 drop all the projected labels and use the working label as the tree decomposition labeling function. Algorithm 1 shows this conversion.
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 $T_{J_Q} = ((I, F), X)$

Lemma 3.1. 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:** To see how Algorithm 1 works, consider the three properties of a tree decomposition:

1. $\bigcup_{i \in I} X_i = V$ by definition of the leaf nodes of $J_Q$.

2. For every edge $(x, y) \in E$, there needs to be an $i \in I$ with $x, y \in X_i$. Once again, all the leaf nodes correspond to a relation $R_j \in \mathcal{R}$. Therefore, for any pair of attributes $x, y$ if some relation $R_j$ contains both $x$ and $y$ there will be some node $v$ such that $L_w(v) = R_j$, so the corresponding node in $T_{J_Q}$ will also have $X_v = R_j$.

3. For all $r, s, t \in I$, if $r$ is on the path from $s$ to $t$ in $T$, then $X_s \cap X_t \subseteq X_r$. This follows due to the definition of when a variable becomes “unnecessary” and is thus projected out and will not be seen in a parent node’s label. A variable $x$ is “unnecessary” at a node $u$ when, other than the subtree rooted $u$, no other node’s working label contains $x$. Therefore, for each node $r$ on the path from $s$ to $t$, if a variable appears in $X_s \cap X_t$ and not in $X_r$, then $x$ must be “unnecessary” along the path from $X_s$ to $X_r$, meaning it does not occur in
$X_t$ by definition of $L_w$, a contradiction since $x \in X_s \cap X_t$. Therefore, the third property holds.

As a result, $T_{jq}$ is a tree decomposition and the width of $T_{jq}$ is $\max_{i \in I} |X_i| - 1 = \max_{v \in V} |L_w(v)| - 1 = k - 1$. 

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 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.

**Lemma 3.2.** [Klo94]. Given a graph $G$ and a tree decomposition $(T = (I, F), X)$ of $G$, for every clique $\{x_1, \ldots, x_n\} \in C$ the tree decomposition has a node $i \in I$ such that $\{x_1, \ldots, x_n\} \subseteq X_i$.

**Lemma 3.3.** 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 $R_j \in \mathcal{R}$.

**Proof:** by construction using Algorithm 2. To show the algorithm works we first show every leaf node of $T'$ has a label containing a $R_j$ for some $R_j \in \mathcal{R}$. Consider if a leaf node $i \in I'$ did not have a $R_j \in \mathcal{R}$ such that $R_j \subseteq X'_i$. Then the node would never be marked because it neither contains a relation nor is it on the path between two nodes contain a relation. So it would have been deleted. Therefore all the leaf nodes of $T'$ contain a relation $R_j \in \mathcal{R}$ in their label.

Now let us check that the three properties of tree decomposition hold:

1. By Lemma 3.2 all the cliques formed by $\mathcal{R}$ are in $X$ and remain in $X'$ by construction. Therefore, for every vertex $u \in V$ there is some clique $R_j \in \mathcal{R}$
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 : R → I’

1: for every relation R_j ∈ R ∪ {R_T} do
2:     Find a node i ∈ I such that R_j ⊆ 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 ∈ 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 | x ∈ X_k, x is marked in X_i, and x is marked in X_j\}
9:     end for
10: end for
11: for every node i ∈ I do
12:     Delete all unmarked labels in X_i.
13:     if X_i == ∅ then
14:         Delete i {If the label is empty, delete the node and corresponding edges}
15:         for all edges e ∈ F containing i do
16:             Delete e
17:         end for
18:     end if
19: end for
20: RETURN (T = (I, F), X), r
that contains \( u \) and there is consequently some node \( i \in I' \) such that \( R_j \subseteq X'_i \), so \( u \in X'_i \).

2. By Lemma 3.2 all edges that are in a clique in \( G \) will be in \( T' \), and since every edge is in some clique (of size 2 at least) every edge of \( G \) is in \( T' \).

3. We need to show that for all \( i, j, k \in I' \) if \( j \) is on the path from \( i \) to \( k \) in \( T' \), then \( X'_i \cap X'_k \subseteq X'_j \). By construction the path between every pair of nodes was marked to contain the marked intersection. Thus, no necessary attribute would have been deleted.

The width of \( T' \) cannot be greater than \( k \) because for any node \( i \in I' \), \( |X_i| \geq |X'_i| \) since the algorithm only deletes from the labels.

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

**Lemma 3.4.** 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 \( k + 1 \).

**Proof:** Consider the output of Algorithm 3 as \( J_Q = (T_Q = (V_Q, E_Q, v_0), L_w, L_p) \). It is a join-expression tree of \( Q \) with join width \( k \). First, \( J_Q \) contains all the relations \( R_j \in \mathcal{R} \) as leaf nodes in \( T_Q \). This is by construction.

Second, the working label of a node \( u \in V_Q \) is the result of the join of the projected labels of the children of \( u \). Since the join expression tree is a tree decomposition with added relational leaf nodes that do not affect any of the tree decomposition properties, the third property of tree decomposition also holds for the working labels of \( J_Q \). First, due to the mark and sweep of Algorithm 3, the working labels do not contain any more vertices than those that appear in their children’s projected label. And a child will not have a node in a project label that will not be in the working label of the parent by construction of the projected label.
Algorithm 3 Tree-Decomposition-to-Join-Expression-Tree(Q,G_Q,T_J_Q)  

Require: A conjunctive query Q, join graph G_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 = 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 ∈ R do  
6: z = newnode(R_j)  
7: V_Q = V_Q ∪ z {create a new node for each relation, these will be the leaf nodes of the join-expression tree}  
8: E_Q = E_Q ∪ (r[R_j], z) {add an edge connecting the relational node to the tree decomposition}  
9: end for  
10: for every node i ∈ V_Q do {Set up working labels}  
11: if i is a leaf node introduced for relation R_j ∈ 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 ∈ V_Q do {Set up projected labels}  
18: if i is a leaf node introduce for relation R_j ∈ R then  
19: L_p(i) = R_j  
20: else if i ≠ v_0 then  
21: L_p(i) = L_w(i) ∩ L_w(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)
Third, the projected label of a node \( u \in V_Q \) does not contain any “unnecessary” vertices. By construction \( L_p(u) \) contains vertices in \( L_w(u) \cap L_w(\text{parent}(u)) \). If an “unnecessary” vertex is in \( L_p(u) \) then it is also needed in \( L_w(\text{parent}(u)) \). But then the vertex is no long “unnecessary” because it belongs in the tree outside of the subtree form at \( u \). Therefore \( L_p(u) \) has no “unnecessary” vertices.

Finally, the working labels of \( J_Q \) are subsets of the labels of the tree decomposition. Therefore the nodes of the join-expression tree that come from the tree decomposition have width no greater than \( k \). The relational nodes also do not have width greater than \( k \). If one did have a width of \( k' > k \) then there would be a clique in \( G_{RV} \) of size \( k' \) and by Lemma 3.2 some node in the tree decomposition must contain this clique. That means the tree decomposition has a treewidth greater than \( k \), a contradiction. So the join-expression tree has a join width of \( k \).

Lemma 3.1 gives us an upper bound on the join width of a query \( Q \) as the treewidth of the join graph of \( Q \) plus one. Conversely, Lemma 3.4 shows that the treewidth plus one over the same join graph is an lower bound for the join width of the query. Therefore, the join width of a query \( Q \) is the treewidth of the join graph of \( Q \) plus one.

Theorem 3.1 offers a graph-theoretic characterization of the power of projection pushing and join reordering. The theorem extends results in [DKV02] regarding rewriting of Boolean conjunctive queries, expressed in the syntax of first-order logic. They explore rewrite rules whose purpose is to rewrite the original query \( Q \) into a first-order formula using a smaller number of variables. Suppose we succeed in rewriting \( Q \) 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 [Var95]. Given a Boolean conjunctive query \( Q \), we would 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 [DKV02] that if $k$ is a positive integer and $Q$ a Boolean conjunctive query, then the join graph of $Q$ has treewidth $k$ iff there is an $L^k$-sentence $\psi$ that is a rewriting of $Q$. Theorem 3.1 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.

### 3.4.2 Bucket Elimination

Unfortunately, we cannot easily turn Theorem 3.1 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 [ACP87]. An alternative strategy to minimizing the width of intermediate results is given by the *bucket-elimination* approach for constraint-satisfaction problems [Dec99], which are equivalent to Boolean project-join queries [KV00]. We now rephrase this approach and extend it to general project-join queries. Assume that we are given an order $x_1, \ldots, x_n$ of the attributes of a query $Q$. We start by creating $n$ “buckets”, one for each variable $x_i$. For an atom $r_i(x_{i_1}, \ldots, x_{i_k})$ of the query, we place the relation $r_i$ with attributes $x_{i_1}, \ldots, x_{i_k}$ in bucket $\max\{i_1, \ldots, i_k\}$. 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, in which $x_i$ 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 of the join 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 pro-
cess 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 [Fre90]. 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 3.2.** The induced width of a project-join query is its treewidth.

This theorem extends the characterization in [DP87, Fre90] for Boolean project-join queries. The proof for the theorem is given in chapter 4.1.

Theorem 3.2 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 [ACP87], it follows from Theorem 3.2 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 [Bou99, RD00, SV01], and use the maximum-cardinality search order (MCS order) of [TY84]. 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).

Given our pentagon example: $\pi_\emptyset \text{edge}(v_1, v_2) \bowtie \text{edge}(v_1, v_3) \bowtie \text{edge}(v_4, v_5) \bowtie \text{edge}(v_3, v_4) \bowtie \text{edge}(v_2, v_3)$, the bucket elimination method would produce the following SQL query:

```
SELECT 1
WHERE EXISTS ( SELECT *
    FROM edge e3 (v4, v5) NATURAL JOIN (...
```
SELECT <DISTINCT> e4.v4, t1.v5
FROM edge e4 (v3, v4) NATURAL JOIN (SELECT <DISTINCT> e2.v5, t3.v3
FROM edge e2 (v1, v5) NATURAL JOIN (SELECT <DISTINCT> e1.v1, e5.v3
FROM edge e1 (v1, v2) NATURAL JOIN edge e5 (v2, v3)) AS t3) AS t1) AS t5);

3.5 Experimental Results

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). The first parameter fixed the order of the query and compared the performance of the optimization as the density of the formulas increased. This tested how, as the structure of the queries change, the methods used scaled with the structural change. The second parameter fixed the density of the query in order to compare how each of the optimizations scales as the order of the formulas increased. This tested how, for a fixed amount of constraint between variables, the optimizations are able to scale with the size of the query. Finally, we perform order scaling on the structured queries to show how the methods scaled when the join width of the query is fixed and the query grows in size.

3.5.1 Density Scaling

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 change, the methods used scale with the structural change. Figure 3.2 shows the median running times (logscale) of each optimization method as we increase the density of the generated 3-COLOR instances. The plot on
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 is 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.

![Figure 3.2: 3-COLOR Density Scaling, Order = 20 – Logscale](image)

Note that at low densities each optimization method improves upon the previous methods. The sparsity of the instances at low densities allows 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 under-constrained and over-constrained regions. The non-Boolean instances demonstrate a similar behavior.

Next, we fixed the order of 3-SAT formulas to 20 and compared the performance of our methods as the density was increased from 1 to 9. Figure 3.3 shows the median
running time of each of the optimizations. As in the 3-COLOR case, the greedy heuristics are only effective in under-constrained problems. As the likelihood that the query is unsatisfiable becomes 1, the greedy heuristics level off. Note that they do not decrease as the density increases as in the case with DPLL and BDD based solvers [CDS+00, CM01, SV01, Bry86]. This is most likely because databases do not use an early exit strategy. So once a join has produced an empty relation, the database still has to process the remaining joins. Bucket elimination, on the other hand, dominates along the entire structural range.

3.5.2 Order Scaling

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 3.4 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 logscale), but
bucket elimination maintains a lower slope. The lower slope means that the exponent of the method is strictly smaller and we have an exponential improvement. Figure 3.5 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 under-constrained and the over-constrained densities, while bucket elimination still finds opportunities for early projection and shows exponential improvement over the other optimizations.

![Figure 3.4: 3-COLOR Order Scaling, Density = 3.0 – Logscale](image)

For 3-SAT, we fixed two densities and compared how they scaled as the order of the formula increased. Figure 3.6 shows the median time in logscale as formulas of density 2.0 increased in size. This density corresponded to (most likely) satisfiable queries. In this case we do see a difference between the greedy heuristics. The early projection shows a slightly smaller slope than the straightforward method produces, and similarly the reordering method shows a similar improvement. But these slight gains are overshadowed by the comparatively drastic gain that bucket elimination makes. The non-Boolean case outlines these gains more distinctly. Figure 3.7 plots in logscale the median running times of the optimizations at density 6.0. This density
corresponds to (most likely) unsatisfiable formulas. Here we see that the greedy heuristics are not able to produce any gains over each other. With not much to use as leverage, they were unable to find a suitable plan for the query. Bucket elimination, on the other hand, once again is still able to exploit some structure in the query and use it to leverage an advantage over the other optimizations.
3.5.3 Color Scaling

The density and order scaling previously shown fixed the number of colors (or literals in a clause). By fixing a query and increasing the number of colors, we in effect increase the size of the database. This allows us to see if the results are an artifact of the small databases, or if our methods do indeed work for large databases. Figure 3.8 shows the results when we fix the \( n \)-COLOR query to a graph of order 5 and density 1.0. The left plot is drawn in logscale and shows that bucket elimination outperforms the other optimizations hands down. Next in performance is early projection. Surprisingly, clause reordering performs the worse, suggesting that the natural order of the query worked slightly better than the one found by the reordering heuristic. The right plot is drawn in loglogscale. Each optimization is a straight line, suggesting they are polynomial in running time. Since the query is fixed and the database is growing, this is what we expect. The main thing to note, however, is that the slopes of the lines are different, which shows that the degree of the polynomial is smaller for bucket elimination than for the other optimizations. This shows that bucket elimination obtains an improvement even as the size of the database increases.
Our next focus was to run order-scaling experiments for the structured queries. These tested how, when the join width of the queries are fixed, the optimizations scale as the query size increases. Figure 3.9 shows the running time (logscale) for the

3.5.4 Structured Queries

Figure 3.9 : 3-COLOR Augmented Path Queries – Logscale
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 because there are 20% fewer vertices to 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 3.10 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 in 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 3.11 and the augmented circular ladder instances in Figure 3.12, 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.
Chapter 4

Tree Decomposition Heuristics

In the previous chapter we explored different heuristical methods for producing a project-join evaluation order for conjunctive queries. Bucket elimination, in conjunction with the MCS variable order, produced the best empirical results. This chapter first focuses more on bucket elimination by including a new proof of the relationship between treewidth and induced width. The chapter then considers possible variable orders (other than MCS) to feed into the bucket elimination method. The chapter wraps up with experimental results and a discussion comparing the performance of these variable orders.

4.1 Induced Width = Tree Width

Bucket elimination is usually analyzed using the notion of induced width [RD00]. Induced width has been shown equal to treewidth, c.f. [Fre90], via a construction from each to k-trees. This section provides a more direct proof using algorithms to transform a treedecomposition of treewidth $k$ to a variable order with induced width $k$ and vice versa.

**Definition 4.1.** Given a graph $G = (V, E)$ and an order of its nodes $o$, the *parent set* of a node $v \in V$ is the set of nodes neighboring $v$ that precedes $v$ in $o$. The size of this parent set is called the *width* of $v$ relative to $o$. The width of the graph along $o$, denoted $w_o$, is the maximum width over all vertices. The *induced graph* of $G$ along $o$, denoted $I_o(G)$, is obtained as follows: going from $i = n$ to $i = 1$, add an edge between the neighbors of $v_i$ that precedes $v_i$ in the order. The induced width of $G$ along $o$, denoted $w_o^*$, is the width of $I_o(G)$ along $o$, while the induced width $w^*$ of $G$ is the
minimum induced width along all orders [RD00].

**Definition 4.2.** 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 \( G \) [RS86].

This section presents an algorithm that takes a graph \( G = (V, E) \) and an order \( o \) to produce a "corresponding" tree decomposition of \( G \). The intuition behind the algorithm is to apply a graph triangulation procedure along the order \( o \) to form the nodes of the tree decomposition. In the algorithm, the following notion will be used. Let \( o : \mathbb{N} \to V \) be bijective function such that \( o(i) \) returns the vertex \( v \) of rank \( i \) in the order \( o \). Similarly \( o^{-1}(v) \) returns the rank of \( v \) in the order. Let \( |V| = n \) and define the parent set of a node \( v \) as \( P(v) = \{ u | (u, v) \in E \cup E_{I_o} \land o^{-1}(u) < o^{-1}(v) \} \), where \( E_{I_o} \) contains the induced edges of \( G \) given \( o \), \( E_{I_o} = \{ (u, v) | \exists y, (u, y) \in E \land (v, y) \in E \land o^{-1}(u) > o^{-1}(y) \land o^{-1}(v) > o^{-1}(y) \land (u, v) \not\in E \} \). The pseudo code is presented in Algorithm 4.

It is also possible to go the other direction, taking in a tree decomposition \((X, T = (I, F))\) for a graph \( G \) and return a "corresponding" order \( o \) of vertices \( V \in G \). The tree from the tree decomposition is undirected, although we can take any node as the root and view it as a directed tree from that root. Given a topological traversal of the tree, the order of first occurrences of each vertex yields a variable order. Let \( |V| = n \) and \( o \) be a queue whose union operator, \( \cup \), appends the variable to the end of the list. The pseudo code is presented in Algorithm 5.
Algorithm 4 Induced-Width-to-Tree-Decomposition($G, o$)

Require: The graph $G = (V, E)$, and an order $o$

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

1: $I = \emptyset$ \{the nodes of the tree\}
2: $F = \emptyset$ \{the edges of the tree\}
3: $X = \emptyset$ \{the labels for the nodes\}
4: $E_{I_o} = \emptyset$ \{induced edges of $G$\}
5: for $i = n$ to 1 do \{Process each vertex of $G$ in reverse order\}
   6: \quad $I = I \cup \{i\}$ \{create a new node for $o(i)$\}
   7: \quad $X_i = P(o(i)) \cup \{o(i)\}$ \{label it with its parent set\}
   8: \quad $X = X \cup \{X_i\}$
   9: \quad for $j = i + 1$ to $n$ do \{add an edge to every disconnected component\}
      10: \quad \quad if there is no path between $j$ and $i$ in the tree $(I, F)$ and $X_j$ contains $o(i)$
      11: \quad \quad \quad $F = F \cup \{(i, j)\}$
      12: \quad \quad end if
      13: \quad end for
   14: \quad for $j, k \in \{1, \ldots, i - 1\}, j \neq k$ do \{connect neighbors of $o(i)$ in $G$\}
      15: \quad \quad if $(o(i), o(j)) \in E \land (o(i), o(k)) \in E \land (o(j), o(k)) \notin E \cup E_{I_o}$ then
      16: \quad \quad \quad $E_{I_o} = E_{I_o} \cup (o(j), o(k))$
      17: \quad \quad end if
      18: \quad end for
   19: end for
20: RETURN $(T = (I, F), X)$

Algorithm 5 Tree-Decomposition-to-Induced-Width($X, T = (I, F)$)

Require: The tree decomposition $(X, T = (I, F))$ for a graph $G = (V, E)$

Ensure: A variable order $o$ for $G$

1: $o = \emptyset$ \{the order of vertices\}
2: $v =$Topological-Sort($T$)
3: for $i = 1$ to $|I|$ do \{for every node in $T$\}
   4: \quad for $u \in X_{v(i)}$ do \{for every vertex in $v(i)$’s label\}
      5: \quad \quad if $u \notin o$ then \{if it is a new node, add it to the order\}
      6: \quad \quad \quad $o = o \cup u$
      7: \quad \quad end if
   8: \quad end for
   9: end for
10: RETURN $o$
Lemma 4.1. Given a graph $G$ and an order of its nodes $o$ where $w_o^* = k$, there is a tree decomposition $T = ((I, F), X)$ of $G$ such that the treewidth of $T$ is $k$.

Proof:

Algorithm 4 constructs such a tree decomposition. By construction, a node is never connected to another node in the same connected component, so the resulting $(I, F)$ forms a tree\(^1\). It is a tree decomposition, because

1. For each $v \in V$, $v \in X_{o(v)} \subseteq X$, and thus $\bigcup_{i \in I} X_i = V$.

2. For every edge $(v, w) \in E$, there needs to be an $i \in I$ with $v, w \in X_i$. W.l.o.g., take $o^{-1}(v) > o^{-1}(w)$. We can see $v \in X_{o(v)}$ by construction, and $w \in X_{o(v)}$ since $o^{-1}(v) > o^{-1}(w)$ and $(v, w) \in E$.

3. For all $i, j, k \in I$, if $j$ is on the path from $i$ to $k$ in $T$, then $X_i \cap X_k \subseteq X_j$. Take an arbitrary vertex $v \in V$. Consider a path $P_{j_0, v} = \langle j_0, j_1, \ldots, j_m \rangle$ in the tree decomposition where $j_0 < j_1 < \ldots < j_m$, $j_0$’s label contains $v$, and $j_m = o^{-1}(v)$ which is the last node in $I$ whose label contains $v$. Such a path always exists due to lines 9-11 of Algorithm 4 which ensures that $j_0$ is connected to $j_m$ through nodes that are between $j_0$ and $j_m$. For every node $j_k$ on the path, the vertex $o(j_k)$ is in $X_{j_{k-1}}$ by construction. If $v$ exists in $X_{j_{k-1}}$, then the edge $(v, o(j_k)) \in E \cup E_{L_o}$. Therefore $v \in X_{j_k}$. By induction, since $v \in X_{j_0}$, then $v$ is in the label of every node in the path. Therefore, for every pair of nodes $i, j$ such that $v \in X_i$ and $v \in X_j$ for some vertex $v$, the path between $i$ and $j$ is a subset of nodes of the two paths $P_{i, v}$ and $P_{j, v}$. Both paths contain $v$ in the label of all its nodes, so $v$ is in the label of every node on the path from $i$ to $j$. Therefore, for all $i, j, k \in I$, if $j$ is on the path from $i$ to $k$ in $T$, then $X_i \cap X_k \subseteq X_j$.

\(^1\)Or a forest if the original $G$ has multiple disconnected components, but we can connect the forest arbitrarily without affecting tree decomposition properties.
Thus $T$ is a tree decomposition on $n$ nodes where each node $i$ contains exactly the induced parent set of a distinct vertex $o(i) \in V$ as well as the vertex $o(i)$ itself. Thus if the induced width of the graph $G$ under the order $o$ is $k$, by definition the maximum size of all the induced parent set, $w^*_o$ is $k$. Thus the maximum size of any label $|X_i| = k + 1$, which makes a treewidth of $k$. 

**Lemma 4.2.** Given a tree decomposition $(X, T = (I, F))$ of a graph $G$ with treewidth $k$, there is an order $o$ of $G$ such that $w^*_o = k$.

**Proof:** We proceed via contradiction. Assume Algorithm 5 returns an order in which $w^*_o > k$, in other words the induced width of some node $v$ is greater than $k$. Then consider the first node of $T$ where $v$ occurs in its label, say $n_v$. For every parent $u$ of $v$, we know $u$ appears no later than $v$ in the tree since we are traversing in topological order. Due to the second property of tree decomposition, at least one node contains both $v$ and $u$, and in fact the first node that contains $v$ will contain $u$ due to the third property of tree decomposition (if a later one contains $u$ and $v$, then $n_v$ is on the path between the first occurrence of $u$, $n_u$, and this node).

Since for every parent of $v$ the first node where $v$ occurs, its parent will also occur, we now consider the induced parents of $v$. Let $y$ be an induced parent of $v$. Because $y$ has a lower rank than $v$ in $o$, the first node that contains $y$, say $n_y$, appears no later than $n_v$ in the tree decomposition. But to be an induced parent, some variable $z$ with higher rank than $v$ must have both $v$ and $y$ as a neighbor. Therefore, $y$ must remain in the tree until $z$ has occurred from the second property of tree decomposition. Then due to the third property $y$ must be contained in all the nodes on the path to $z$ including the ones that contain $v$. Therefore $n_v$ contains its entire induced parent set including itself, which means the treewidth of $T$ is greater than $k$. A contradiction, and therefore the order has induced width $w^*_o = k$. 

**Theorem 4.1.** Given a graph $G$, the induced width $w^*$ of $G$ is equal to the treewidth of $G$. 

Proof: From Lemma 4.1 we know that if the induced width \( w^* = k \), then there is an order \( o \) of \( V \in G \) that gives \( w^*_o = k \), and from Algorithm 4 we can obtain a tree decomposition of width \( k \). So \( tw(G) \leq k \). Analogously, from Lemma 4.2, we know that if there exists a tree decomposition of width \( k \), then there is a variable order \( o \) from Algorithm 5 such that \( w^*_o = k \). Thus \( w^* \leq w^*_o = k \). Combining both facts gives that the induced width of \( G \) is equal to the treewidth of \( G \), \( w^* = tw(G) \).  

4.2 Variable Order Heuristics

The bucket elimination method discussed in Chapter 3.4.2 used a heuristical variable order called Maximum Cardinality Search (MCS) developed by Tarjan and Yannakakis [TY84]. This section introduces two new variable order heuristics, each a variant on a lexicographic breadth first search.

Graph triangulation forms the backbone of all three variable heuristics. A graph is called triangulated if every cycle of length at least four contains a chord. Koster, Bodlaender, and van Hoesel [KBvH01] show that finding a treewidth of a graph \( G \) is equivalent to finding a triangulation of \( G \) with minimal clique size. Finding a triangulation of minimal clique size is NP-hard, but the clique size of any triangulation of \( G \) now forms an upper bound for the treewidth of \( G \). Using this fact, a heuristic for finding a good tree decomposition of \( G \) can simply triangulate \( G \). But how do we triangulate a graph?

Given a graph \( G = (V, E) \), a vertex \( v \in V \) is simplicial if the neighbors of \( v \) induce a complete subgraph of \( G \). An order \( o \) of the vertices is called a perfect elimination scheme if \( v_i \) is a simplicial vertex of the induced subgraph of \( G \) restricted to the vertices \( v_i, \ldots, v_n \), for all \( i = 1, \ldots, n \). Fulkerson and Gross [FG65] show us how we might triangulate a graph by proving that a graph if triangulated if and only if it has a perfect elimination scheme.

The three variable heuristics in this chapter all attempt to build a perfect elimination scheme of the given graph \( G \). Given an initial vertex, they each compute
the variable order incrementally. All three algorithms guarantee that the elimination scheme is perfect if and only if $G$ is triangulated. An analysis of these heuristics in their ability to determine accurate treewidths can be found in [KBvH01].

4.2.1 Maximum Cardinality Search

Given a graph $G = (V, E)$ where $|V| = n$, the maximum cardinality search (MCS) iterates $n$ times. At each iteration, MCS will pick a vertex and place it in into the next available position in the ordering. Initially, the algorithm is given a vertex to start with and at each subsequent iteration, it picks the vertex that has the highest connectivity with the vertex already chosen, breaking ties randomly. In other words, MCS picks the vertex with the greatest number of neighbors in the set of vertices already picked. This algorithm runs in $O(n + m')$ where $m' = |E| + |E_{I_o}|$ is the number of edges and induced edges based on this ordering [KBvH01].

4.2.2 Lexicographic Breadth First Search

First used as a recognition algorithm in [RTL76], the lexicographic breadth first search algorithm was extended in [KBvH01] to find a vertex order in which to triangulate a graph. Two variants of the lexicographic breadth first search algorithm were developed, the first focused on finding a perfect elimination scheme (LEX_P) and the second with minimizing the triangulation of the given graph $G$ (LEX_M). Algorithm 6 shows the pseudo code for the LEX_P variant. The algorithm labels the vertices with their already ordered neighbors using the positions of those neighbors in decreasing order. At each iteration, the algorithm picks the vertex with the lexicographic highest label. This algorithm runs in $O(n + m')$ where $m' = |E| + |E_{I_o}|$ is the number of edges and induced edges based on this ordering [KBvH01].

The second variant of the lexicographic breadth first search, LEX_M, can be found in Algorithm 7. The focus of the heuristic is to find a minimal triangulation of $G$. This minimization step forces the algorithm to have a more complicated labeling
Algorithm 6 LEX_P(G = (V, E), v_0 ∈ V) [KBvH01]

Require: The graph $G = (V, E)$ and a starting vertex $v_0$
Ensure: A variable order $o$

1: $E_t = \emptyset$ \{induced edges\}
2: $o = \emptyset$ \{the order of vertices\}
3: $S = V$ \{the set of unordered vertices\}
4: for $i = |I|$ to 1 do \{for every node in T\}
5: \hspace{1cm} if $i == |V|$ then
6: \hspace{2cm} $u = v_0$
7: \hspace{1cm} else
8: \hspace{2cm} $u = \text{arg max}_{v \in S} \text{label}(v)$ \{u is a vertex with lexicographic largest label\}
9: \hspace{1cm} end if
10: $o[i] = u$ \{add u to the ordering\}
11: for $j, k \in \{i, \ldots, |V|\}, j \neq k$ do
12: \hspace{1cm} if $(o[i], o[j]), (o[i], o[k]) \in E \land (o[j], o[k]) \notin E \cup E_t$ then
13: \hspace{2cm} $E_t = E_t \cup (o[j], o[k])$ \{add induced edge\}
14: \hspace{1cm} end if
15: end for
16: $S = S \setminus \{u\}$
17: for $w \in \text{Neighbor}(u) \cap S$ do
18: \hspace{1cm} label($w$) = label($w$) $\cup \{i\}$ \{add i to the label of all unordered neighbors\}
19: end for
20: end for
21: RETURN $o$
function. Now, instead of just labeling neighbors with already ordered vertices, the algorithm labels a vertex that has a path to an ordered vertex through unordered vertices with lower labels. This extra labeling procedure brings the running time up to $O(nm')$ [KBvH01].

Algorithm 7 LEX\_M($G = (V, E), v_0 \in V$) [KBvH01]

**Require:** The graph $G = (V, E)$ and a starting vertex $v_0$

**Ensure:** A variable order $o$

1: $E_i = \emptyset$ \{induced edges\}
2: $o = \emptyset$ \{the order of vertices\}
3: $S = V$ \{the set of unordered vertices\}
4: for $i = |I|$ to 1 do \{for every node in $T$\}
5: \hspace{1cm} if $i == |V|$ then
6: \hspace{2cm} $u = v_0$
7: \hspace{1cm} else
8: \hspace{2cm} $u = \arg \max_{v \in S} \text{label}(v)$ \{u is a vertex with lexicographic largest label\}
9: \hspace{2cm} end if
10: $o[i] = u$ \{add $u$ to the ordering\}
11: for $j, k \in \{i_1, \ldots, |V|\}, j \neq k$ do
12: \hspace{1cm} if $(o[i], o[j]), (o[i], o[k]) \in E \land (o[j], o[k]) \notin E \cup E_t$ then
13: \hspace{2cm} $E_t = E_t \cup (o[j], o[k])$ \{add induced edge\}
14: \hspace{1cm} end if
15: end for
16: $S = S \setminus \{u\}$
17: for $w \in S : \exists$ path $\{u = v_1, \ldots, v_{k+1} = w\}$ in $G$
\hspace{2cm} with $v_j \in S$ and $\text{label}(v_j) < \text{label}(w)$ for $j = 2, 3, \ldots, k$ do
18: \hspace{1cm} $\text{label}(w) = \text{label}(w) \cup \{i\}$ \{add $i$ to the label of all unordered neighbors\}
19: end for
20: end for
21: RETURN $o$

4.3 Experimental Results

Using the same experimental setup as in Chapter 3.5, we varied two parameters of the project-join queries to test the scalability of the different variable heuristics. Again, we measured and report the query execution time. The first parameter fixed
the order of the query and compared the performance of the optimization as the density of the formulas increased. This tested how, as the structure of the query changes, the methods used scaled with the structural change. The second parameter fixed the density of the query in order to compare how each of the optimizations scaled as the order of the formulas increased. This tested how, for a fixed number of constraints between variables, the optimizations are able to scale with the size of the query. Finally, we performed order scaling on the structured queries to show how the methods scale when the join width of the query is fixed and the query grows in size.

4.3.1 Density Scaling

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 scale with the structural change. Figure 4.1 shows the median running times (logscale) of each heuristic applied to the bucket elimination method as we increase the density of the generated 3-COLOR instances. The plot on the left shows the Boolean query, and the right one considers the non-Boolean case with 20% of the vertices in the target schema. The shapes of the curves are roughly the same. We see a relatively fixed size difference between the performances of the heuristics, but they all scale similarly over the structural change. MCS leads the pack in its performance, but LEX_P is very competitive.

Next, we fixed the order of 3-SAT formulas to 20 and compared the performance of our methods as the density was increased from 1 to 9. Figure 4.2 shows the median running time of each of the optimizations (note that it is not logscale). As in the 3-COLOR case, the basic shapes of the curves are roughly the same. We do see a difference in performance, particularly in the non-Boolean case. MCS performs the strongest, followed by LEX_P and then by LEX_M. However, in terms of scalability the plot does not provide much to differentiate between them.
4.3.2 Order Scaling

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 4.3 shows the
running times (logscale) for the different heuristics at density 3.0 as order is scaled from 10 to 35. Here we see LEX\_P take over as the overall performer for the queries in both the Boolean and non-Boolean case. Figure 4.4 shows the experiment for density 6.0 as order is scaled from 15 to 30. Here we see the differences between the heuristics become more pronounced. MCS has once again taken over all as the leader, with a sizable gain over the lower orders. As the order of the formula increases, though, we see that LEX\_P is able to find places to increase performance and starts matching the MCS execution times. Throughout the range of queries, LEX\_M never becomes as viable an option as the other two heuristics.

For 3-SAT, we fixed two densities and compared how they scaled as the order of the formula increased. Figure 4.5 shows the median time in logscale as formulas of density 2.0 increased in size. This density corresponded to (most likely) satisfiable queries. In this case we do see a difference in slope between the variable order heuristics. MCS and LEX\_P both show an exponential improvement over LEX\_M, and in the non-Boolean case MCS also achieves the same gain over LEX\_P. Conversely, Figure 4.6 plots in logscale the median running times of the optimizations at density 6.0. This
density corresponds to (most likely) unsatisfiable formulas. Here we do not see the exponential improvement as found in the low density, and all the heuristics perform basically the same.
4.3.3 Structured Queries

Our next focus was to run order-scaling experiments for the structured queries. These tested how, when the join width of the queries is fixed, the optimizations scale as the query size increases. Figure 4.7 shows the running time (logscale) for the optimization methods as the augmented path instances scaled. In the Boolean case, MCS
and \( \text{LEX}_\text{P} \) are both able to create queries that correspond with the graph’s low treewidth. \( \text{LEX}_\text{M} \), on the other hand seems to stumble around, sometimes finding good solutions, other times finding terrible solutions. In the non-Boolean case, the treewidth of the queries is no longer fixed but scale with the size of the query. All of the heuristics, in this case, have a similar slope again, but MCS outperforms both of
the other heuristics and LEX_M still unreliably picks a good variable order. Figure 4.8 shows the running time (logscale) for the methods when applied to the ladder graph instances. Again, both the MCS and LEX_P algorithm find variable orders that take advantage of the queries fixed treewidth in the Boolean case, and again LEX_M has a jagged edge to its curve.

As we move to the augmented ladder instances in Figure 4.9 and the augmented circular ladder instances in Figure 4.10, the differences between the optimization methods start to become more stark, especially in the non-Boolean queries. The Boolean augmented circular ladder instances are the first fixed treewidth instances that the LEX_P heuristic was not able to take advantage of. The MCS heuristic, on the other hand, never waivers in its ability to find a good execution plan. Overall we find that throughout both the random and structured queries, LEX_P and MCS both outperform LEX_M, and MCS is the most viable of the three heuristics.
Chapter 5

Conclusions

In this thesis, we demonstrate progress in moving structural query optimization from theory, as developed in [CR98, GLS99, KV00, DKV02], to practice, by using techniques from constraint satisfaction. In particular, we demonstrate 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. These improvements show that the NP-hardness results at the heart of structural query optimization theory need not be considered an insurmountable barrier to the applicability of these techniques. We also show that when the query is non-Boolean but contains free variables, our techniques can make use of those variables to obtain a speed up. Next we demonstrate that MCS provides a solid variable order for the bucket elimination method, but that other variable orders can provide some competitive results. Overall, we have shown both theoretically and empirically that the bucket elimination method provides a solid backbone for finding good plans for queries with a large number of joins.

Current benchmarks do not contain much testing for queries with a large number of joins. For example, most of the queries in TPC-R\(^1\) contain two or three relations, so the effectiveness of the bucket elimination approach is not very promising. But there are a few queries, for example the National Market Share query (Q8) in TPC-R, which contain up to eight relations joined together with most of the attributes available for early projection. Queries such as these could use the bucket elimination

\(^1\)http://www.tpc.org/tpcr/default.asp
method to try to obtain an improvement in running time. Overall, though, current benchmarks do not challenge the scalability aspect of cost-based optimizations.

This work provides 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 [YLUGM99]. This would require extending the methods to handle the need to evaluate certain subgoals before other ones. Second, further ideas should be explored and our methods should be extended using the theory of structural query optimization, e.g., semijoins [WY76] and hypertree width [GLS99], using methods from constraint satisfaction, e.g., mini-buckets [Dec97], using methods from symbolic model checking, e.g., partitioning techniques [CCJ+01], and from using techniques in algorithmic graph theory, e.g., treewidth approximation [Bod93]. Third, heuristic approaches to join minimization (as in [CM77]) should also be explored. Since join minimization requires evaluating a conjunctive query over a canonical query database [CM77], the techniques in this paper should be applicable to the minimization problem, cf. [KS02]. Fourth, structural query optimization needs to be combined with cost-based optimization, which is the current 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. Further work also needs to examine effective tie breakers. For example, if the query contains no free variables, the methods right now would perform arbitrarily, but an effective tie breaker could optimize for something other than the arity of intermediate relations. Finally, there is the question of how our optimization technique can be integrated into the framework of rule-based optimization [Fre87].
Bibliography


