SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Tuning Linear Programming Solvers for Query Optimization

Sarra Ben Mohamed

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Tuning Linear Programming Solvers for Query Optimization

Anpassung von Linear Programming Solvern für Anfrageoptimierung

Author:	Sarra Ben Mohamed
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Altan Birler
Submission Date:	15/10/2023

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15/10/2023 Sarra Ben Mohamed

Acknowledgments

Abstract

Contents

1 Introduction

Our aim with this project is to investigate and compare different methods and techniques to solve small linear programming problems representing among others the problem of cardinality estimation. A way to estimate realistic and useful upper bounds of query sizes is through linear programming. Studies have shown that cardinality estimation is the major root of many issues in query optimization [Ngo22], which is why we want a practical estimate to choose the best from data plans to run efficient queries. For this purpouse, we will introduce a formal description of the cardinality estimation problem, represent it in the form of a packing linear programming problem with the intention of maximizing the size of the query under some constraints. The result is hundreds of relatively small LP that we collect in datasets and solve them with different methods and algorithms. We then draw conclusions based on the results of our experiments, benchmarks and the previous work done on similar packing LP problems. This should guide us into constructing a thorough analysis of the particularities of these LP problems, what's unique about their structure and if their solution process follows any patterns. We then discuss and draw hypotheses on the ways this analysis can be exploited to further optimize the solution process: which methods or combination of methods deliver the best time and memory complexity.

2 Related work

2.1 Background

In this chapter we talk about optimization, in particular the field of linear programming, we focus on the most widely used algorithms to tackle this problem, and we present some use cases and benchmarks for this technique.

In the pipeline of query execution, cardinality estimation serves as a cornerstone for the optimization process. Cardinality, defined as the number of tuples in the output, plays a pivotal role in the selection of an optimal query plan. Modern Database Management Systems (DBMSs) often rely on cost-based Query Optimizers to make this selection. For example, the SQL Server Query Optimizer [Mic23] employs a cost-based approach, aiming to minimize the estimated processing cost of executing a query.

The cost estimation is influenced by two primary factors: the cardinality of the query plan and the algorithmic cost model dictated by the operators used in the query. The former serves as an input parameter for the latter, creating a dependency between the two. Enhanced cardinality estimation can lead to more accurate cost models, which in turn results in more efficient query execution plans.

2.1.1 Linear Programming

Linear programming or optimization is a mathematical modeling technique in which a linear function (called the objective function) $f(\mathbf{x}) = \sum_{j=1}^{n} c_j x_j$ is maximized or minimized when subject to a set of constraints

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m
\end{aligned}$$

Where: x_j are called decision variables for $j = 1, 2, \ldots, n$. c_j, a_{ij} and b_i are constants, and they constitute the coefficient vector **c**, the constraint matrix **A** and the right-hand-side vector **b** respectively. There are m constraints.

The LP problem class that we are dealing with is called the packing LP problem. Additionally it is a special instance where:

- c, the vector of the variable coefficients in the objective function, is a vector of all ones
- b , or the right hand side vector, is a vector of all ones

Our specific problem is then expressed as follows:

$$\begin{aligned}
&\text{Maximize} \quad && \sum_{j=1}^{n} x_j && \\
&\text{subject to} && && \\
& && \sum_{j=1}^{n} a_{ij} x_j \leq 1, && i = 1, \dots, m \\
& && x_j \geq 0, && j = 1, \dots, n
\end{aligned} \tag{2.1}$$

This specific class of LPs has a simple structure that we can exploit, see Chapter 3, to further optimize our implementation.

2.1.2 The Standard Simplex Algorithm

The algorithm

In this subsection we will present the most widely used algorithm for solving LP problems. We have implemented our version of this algorithm in the C++ language and we use it, among others, to solve our dataset. To be approachable by the simplex algorithm, the LP 2.2 needs to be cast in a computational form, that fulfills the requirement of the constraint matrix having to have full row rank and only equality constraints are allowed. To convert the inequalities to equations, we introduce slack variables $s_1, s_2, \dots, s_m$:

$$
\begin{aligned}
\text{Maximize} \quad & z = \sum_{j=1}^{n} x_j \\
\text{subject to} \quad & \sum_{j=1}^{n} a_{1j} x_j + s_1 = 1 \\
& \sum_{j=1}^{n} a_{2j} x_j + s_2 = 1 \\
& \vdots \\
& \sum_{j=1}^{n} a_{mj} x_j + s_m = 1 \qquad (2.2) \\
& x_1, x_2, \ldots, x_n, s_1, s_2, \ldots, s_m \geq 0
\end{aligned}
$$

A simple packing LP in tabular form would look like this:

	a	b	c	d	s_1	s_2	RHS
z	-1	-1	-1	-1	0	0	0
s_1	a_{11}	a_{12}	a_{13}	a_{14}	1	0	1
s_2	a_{21}	a_{22}	a_{23}	a_{24}	0	1	1

At each iteration of the simplex method, we have a basic feasible solution represented by a dictionary.

Example Dictionary

$$
\begin{aligned}
x_5 &= 10 - x_1 - 2x_2 \\
x_6 &= 20 - 2x_1 - 3x_2 \\
x_7 &= 5 - x_1 - x_3 \\
z &= 5x_1 + 4x_2
\end{aligned}
$$

In this dictionary:

- x_5, x_6, x_7 are basic variables.
- x_1, x_2, x_3 are non-basic variables.
- The objective function value is given by the equation for z.

To perform an iteration, we select an entering variable (e.g., x_1) and a leaving variable (e.g., x_6). We then pivot to obtain a new dictionary.

2.1.3 Pivoting

Pivoting on x_1 entering and x_6 leaving, we get a new dictionary.

We call this a feasible dictionary[Chv83], or a feasible tableau. This is apparent, since all values in the RHS are positive, and the constraint matrix A also has positive coefficients.

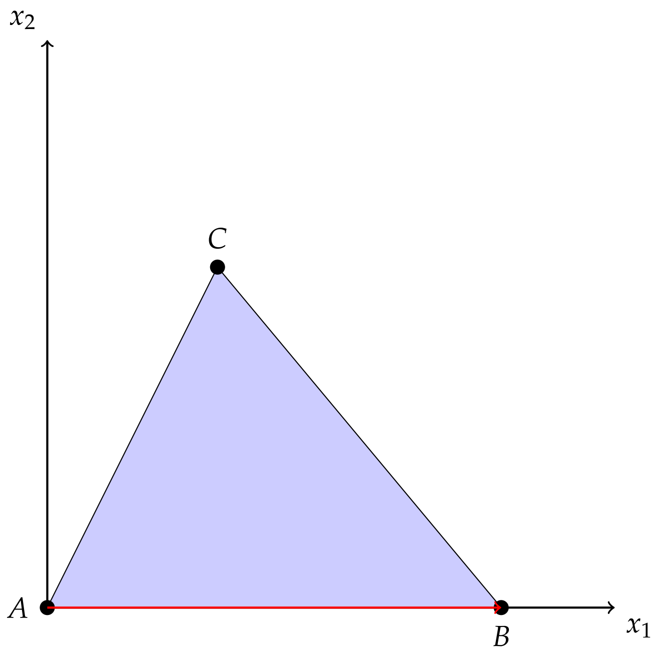
- feasible dictionaries?
- just write the algorithm in abstarct way, in the approach chapter write it in the specific way I implemented it (Bland's rule, zero tolerance, ...)
- the grand strategy of the simplex method is that of successive improvements
- decision variables vs. slack variables
- A maximization problem is optimized when the slack variables are "squeezed out," maximizing the true variables' effect on the objective function. Conversely, a minimization problem is optimized when the slack variables are "stuffed," minimizing the true variables' effect on the objective function.
- feasibility, boundedness,
- largest coefficient rule vs. largest increase rule.
- the problem of stalling, degeneracy
- Bland's rule guarantees termination.

The complexity

The Simplex algorithm for linear programming has an exponential worst-case time complexity, which we denote by $O(2^n)$. For packing linear programs, the worst-case time complexity of the Simplex algorithm remains exponential, even though there exists polynomial time implementations for it. [Sti10].

Graphical interpretation

The simplex algorithm can be understood geometrically as a method to navigate the vertices (corners) of a polytope defined by the feasible region of a linear programming problem. The algorithm starts at an initial vertex and moves along the edges of the polytope to vertices with better objective values until the optimal solution is reached.



In the above figure, the shaded region represents the feasible region of a linear programming problem. The simplex algorithm starts at vertex A and moves to vertex B because B offers a better objective value. The algorithm would continue navigating the vertices until the optimal solution is found.

- The simplex algorithm only evaluates the objective function at the vertices of the feasible region.
- At each step, the algorithm selects an adjacent vertex with a better objective value and moves to it.
- The algorithm terminates when it reaches a vertex where all adjacent vertices have worse objective values, indicating an optimal solution.

2.1.4 The Revised Simplex Algorithm

As explained in the book [Chv83]. while Standard Simplex algorithm maintains and updates the entire tableau in its dense form, which includes both basic and non-basic variables, at each iteration. In contrast, the Revised Simplex algorithm focuses on tracking only the basic variables and their corresponding basis matrix B, thereby reducing memory requirements. The Revised Simplex algorithm often employs the product form update method to efficiently update the inverse of the basis matrix B^{-1} without having to recompute it from scratch, making it computationally more efficient for large-scale problems. Finally, we primarily use the Compressed Column Representation (CCR) of the, which greatly benefits the performance of the implementation.

Algorithm 1 Revised Simplex Algorithm

```
1. Input: A feasible basic solution, $B$, $c$, $A$, and $b$
2. Output: Optimal solution or a certificate of unboundedness
3. Initialize $B^{-1}$, the inverse of the basis matrix $B$
4. While True:
       Step 1: Solve the system $yB = c_B$ (BTRAN)
       Step 2: Choose an entering column. This may be any column a of $A_N$ such
         that $ya$ is less than the corresponding component of $c_N$. If there is no such
         column, then the current solution is optimal. In other words: Choose first $j$
         such that $c_j - yA_j > 0$ then $a = A_j$ is the enterig column.
       Step 3: Solve the system $Bd = a$ (FTRAN)
       Step 4: Let $x_B^* = B^{-1}b$ the current basic variables' values. Find the largest
         $t$ such that $x_B^* - td \geq 0$ if there is no such $t$, then the problem is
         unbounded; otherwise, at least one component of $x_B^* - td$ equals zero and
         the corresponding variable is leaving the basis.
       Step 5: Set the value of the entering variable at $t$ and replace the values $x_B^*$
         of the basic variables by $x_B^* - td$. Replace the leaving column of B by the
         entering column, and in the basis heading, replace the leaving variable by
         the entering variable.
5. Return Optimal solution $B^{-1}b$
```

The product form update method

We will discuss the PFI, introduced by George Dantzig [DO54].

Data structures

Compressed Storage Formats: We use the CSC format to store sparse matrices. These formats store the non-zero values, along with their corresponding row and column indices, in a compact way. This reduces memory usage and speeds up operations on sparse matrices

```
struct CCRMatrix {
    float *values;  // Non-zero values in the matrix
    int *rowIdx;  // Row indices corresponding to the non-zero values
    int *colPtr;  // Points to the index in ‘values‘ where each column starts
};
```

2.1.5 Cardinality Estimation

As previously discussed, accurate and reliable cardinality estimates are crucial in achieving faster query execution times. The objective is to develop a Linear Programming (LP) solver designed specifically for cardinality estimation. This solver aims to maximize a cost function that represents the upper bound of the output size, optimizing for both time and memory complexity. The performance of this solver will be evaluated using the Query-per-Hour Performance Metric (QphH@Size).

To set the stage for our implementation, we focus on the problem of upper-bounding the cardinality of a join query Q. We begin by considering the worst-case join size and then proceed to add some constraints that keep some of our main variables in check. The motivation behind this work is to frame the problem of estimating the size of a multi-join query as a packing linear programming problem.

Scenario

To elucidate the core concepts, suppose we have two relation R and S with attributes

$$Q(a,b,c) = R(a,b) \bowtie S(b,c)$$

where we denote the sizes of the relations as $|R|$ and $|S|$ respectively. It is easy to see that the largest possible output is $|R| \cdot |S|$, which occurs when the join behaves like a cartesian product, i.e. have a selectivity equals to 1. So, this is the worst-case upper bound.

Now the maximum sizes of these relations (i.e. the number of tuples, in our case pairs) depend on the variables a, b and c, their respective types and domains. It can also be affected by te nature of the data or business rules.

If there are constraints on the domains of the attributes, the join size can be limited. For example:

- If B in R and B in S have domain constraints such that they can only take on k distinct values, then the maximum number of join results for a single value of B is $\frac{|R|}{k} \times \frac{|S|}{k}$ (assuming uniform distribution). Therefore, the worst-case join size is $k \times \left(\frac{|R|}{k} \times \frac{|S|}{k}\right) = \frac{|R| \times |S|}{k}$.
- If there are foreign key constraints, such as B in S being a foreign key referencing B in R, then each tuple in S can join with at most one tuple in R. This means the worst-case join size is $\min(|R|, |S|)$.

If there are multiple constraints, they can be combined to provide a tighter bound on the join size. For instance, if there's both a domain constraint and a foreign key constraint, the worst-case join size would be the minimum of the sizes derived from each constraint.

We start with the inequality 2.3. Applying the natural logarithm to both sides yields 2.4. We then rename the variables, simplifying the inequality to 2.5. Normalizing by dividing both sides by r', we obtain 2.6. This leads us to the objective function for our packing LP problem.

$$|a| \cdot |b| \leq |R| \tag{2.3}$$

$$\ln |a| + \ln |b| \leq \ln |R| \tag{2.4}$$

$$a' + b' \leq r' \tag{2.5}$$

$$\frac{1}{r'}a' + \frac{1}{r'}b' \leq 1 \tag{2.6}$$

$$\text{maximize } a' + b' + c' + d' \quad \text{s.t.} \quad \frac{1}{r'}a' + \frac{1}{r'}b' \leq 1 \tag{2.7}$$

And in this simple abstracted way we get a sample packing LP from our dataset.

Variables

Objective

Constraints

2.2 Previous Work

Here we will discuss alternative approaches that are superseded by my work.

2.2.1 Comparative studies of different update methods

We will focus on one study [HH15].

2.2.2 Other techniques

The primal simplex method starts from a trial point that is primal feasible and iterates until dual feasibility. The dual simplex method starts from a trial point that is dual feasible and iterates until primal feasibility. ALGLIB implements a three-phase dual simplex method with additional degeneracy-breaking perturbation:

- Forrest-Tomlin updates for faster LU refactorizations
- A bound flipping ratio test (also known as long dual step) for longer steps
- Dual steepest edge pricing for better selection of the leaving variable
- Shifting (dynamic perturbations applied to cost vector) for better stability

3 Tuning Linear Programming Solvers for Query Optimization

This is the body

3.1 Proposal

3.2 Experimental Design

3.2.1 Analysis of dataset properties

In this subsection we will conduct an analysis of our dataset properties. What are the particularites of the structure of these LP problems, is their any patterns in their solution process. This anaylsis is based on observing the statistical results we obtained from running different solvers on these problems. This will later provide us with insight regarding optimization of these problems. TPC-H is a Decision Support Benchmark The TPC-H is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size)

Mention zero tolerances: A zero tolerance epsilon2 saefguards against divisions by extremely small numbers, which tend to produce the most dangerous rounding errors, and may even lead to degeneracy. diagonal entry in eta matrix should be fairly far from otherwise (in our experiment) degeneracy.

3.2.2 Dataset Structure

Our dataset stucture: as opposed to what the linear programming research has dealt with, which is very large problems, we are dealing with hundreds of small problems. These are represented in the revised simplex algorithm by sparse matrices but not as sparse as it would have been if the problem was large, small matrices that are not small enough to be dense. (they still have quite a number of non-zeroes).

Algorithm 2 Tableau Simplex Algorithm

```
Input: Packing LP maximisation problem in computational form
Output: Optimal value $z$
Step 1: Pricing: Find pivot column, or entering variable using Bland's rule
    $enteringVars \leftarrow$ findPivotColumnCandidates()
    if no entering variable found then
        print "Optimal value reached."
        return $z$
    end if
    $pivotColumn \leftarrow enteringVars[0]$
Step 2: Find pivot row, or leaving variable using the ratio test
    $pivotRow \leftarrow$ findPivotRow($pivotColumn$)
    if no leaving variable then
        print "The given LP is unbounded."
        return $\infty$
    end if
Step 3: Update the tableau using pivotting and update the objective function value
    doPivotting($pivotRow, pivotColumn, z$)
Goto Step 1
```

3.3 Analysis

Some metrics:

- number of iterations
- runtime
- number of loops ?
- for matrix : number of columns and rows, nonzeros and density

3.4 Results

All the following results have been obtained on a personal computer with AMD 4000 series RYZEN, 16GB RAM running Ubuntu. Using the following settings:

- Presolve techniques are not used
- scaling techniques are not used
- The computed optimal solutions have been validated using the scipy python library.

4 Evaluation

4.1 Setup

4.1.1 Evaluation metrics

4.1.2 Evaluation baselines

4.2 Results

4.3 Discussion

5 Conclusion

List of Figures

List of Tables

Bibliography

[Chv83] V. Chvátal. *Linear programming*. Macmillan, 1983.

[DO54] G. B. Dantzig and W. Orchard-Hays. “The product form for the inverse in the simplex method.” In: *Mathematical Tables and Other Aids to Computation* (1954), pp. 64–67.

[HH15] Q. Huangfu and J. J. Hall. “Novel update techniques for the revised simplex method.” In: *Computational Optimization and Applications* 60 (2015), pp. 587–608.

[Mic23] Microsoft. *Cardinality Estimation (SQL Server)*. Accessed: Sep 19th. 2023. URL: `https://learn.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver16`.

[Ngo22] H. Q. Ngo. “On an Information Theoretic Approach to Cardinality Estimation (Invited Talk).” In: *25th International Conference on Database Theory (ICDT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[Sti10] W. Stille. “Solution techniques for specific bin packing problems with applications to assembly line optimization.” PhD thesis. Technische Universität, 2010.