Solving the Join Ordering Problem via Mixed Integer Linear Programming

Immanuel Trummer
itrummer@cornell.edu
Cornell University

Christoph Koch
christoph.koch@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

We transform join ordering into a mixed integer linear program (MILP). This allows to address query optimization by mature MILP solver implementations that have evolved over decades and steadily improved their performance. They offer features such as anytime optimization and parallel search that are highly relevant for query optimization.

We present a MILP formulation for searching left-deep query plans. We use sets of binary variables to represent join operands and intermediate results, operator implementation choices or the presence of interesting orders. Linear constraints restrict value assignments to the ones representing valid query plans. We approximate the cost of scan and join operations via linear functions, allowing to increase approximation precision up to arbitrary degrees. We integrated a prototypical implementation of our approach into the Postgres optimizer and compare against the original optimizer and several variants. Our experimental results are encouraging: we are able to optimize queries joining 40 tables within less than one minute of optimization time. Such query sizes are far beyond the capabilities of traditional query optimization algorithms with worst case guarantees on plan quality. Furthermore, as we use an existing solver, our optimizer implementation is small and can be integrated with low overhead.

Keywords

Query optimization; integer linear programming

1. INTRODUCTION

From the developer's perspective, there are two ways of solving a hard optimization problem on a computer: either we write optimization code from scratch that is customized for the problem at hand or we transform the problem into a popular problem formalism and use existing solver implementations. In principle, the first approach could lead to more efficient code as it allows to exploit specific problem properties. Also, we do not require a transformation that might blow up the size of the problem representation. In practice however, our customized code competes against mature solver implementations for popular problem models that have been fine-tuned over decades [5], driven by a multitude of application scenarios. Using an existing solver reduces the amount of code that needs to be written and we might obtain desirable features such as parallel optimization or anytime behavior (i.e., obtaining solutions of increasing quality as optimization progresses) automatically from the solver implementation. It is therefore in general advised to consider and to evaluate both approaches for solving an optimization problem.

We apply this generic insight to the problem of database query optimization. For the last thirty years, the problem of exhaustive query optimization, more precisely the core problem of join ordering and operator selection [27], has typically been solved by customized code inside the query optimizer. Modern query optimizers are the result of thousands of man years worth of work [19]. The question arises whether this development effort is actually necessary or whether we can transform query optimization into another popular problem formalisms and use existing solvers. We study that question in this paper.

We transform the join ordering problem into a mixed integer linear program (MILP). We select that formalism for its popularity. Integer programming approaches are currently the method of choice to solve thousands of optimization problems from a wide range of areas [21]. Corresponding software solvers have sometimes evolved over decades and reached a high level of maturity [5]. Commercial solvers such as Cplex[1] or Gurobi[2] are available for MILP as well as open source alternatives such as SCIP[3].

Those solvers offer several features that are useful for query optimization. First of all, they possess the anytime property: they produce solutions of increasing quality as optimization progresses and are able to provide bounds for how far the current solution is from the optimum. Chaudhuri recently mentioned the development of anytime algorithms as one of the relevant research challenges in query optimization [8]. Mapping query optimization to MILP immediately yields an algorithm with that property (note that recently proposed anytime algorithms for multi-objective query optimization [32] are not applicable to traditional query opti-

SIGMOD'17 May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064039

[1]http://www.cplex.com
[2]http://www.gurobi.com/
[3]http://scip.zib.de/

mization). Second, MILP solvers already offer support for parallel optimization which is an active topic of research in query optimization as well [13, 36, 28, 34]. Finally, the performance of MILP solvers has improved (hardware- independently) by more than factor 450,000 over the past twenty years [5]. It seems entirely likely that those advances can speed up query optimization as well (and anticipating our experimental results, we find indeed classes of query optimization problems where a MILP based approach treats query sizes that are illusory for prior exhaustive query optimization algorithms).

In summary, by connecting query optimization to integer programming, we benefit from over sixty years of theoretical research and decades of implementation efforts. Even better, having a mapping from query optimization to MILP does not only enable us to benefit from past research but also from all future research and development advances that originate in the fruitful area of MILP. Performance improvements have been steady in the past [5] and, as several major software vendors compete in that market, are likely in the future as well.

Given that integer programming transformations have been proposed for many optimization problems that connect to query optimization [1, 2, 11, 26, 37], it is actually surprising that no such mapping has been proposed for the join ordering problem itself so far. There are even sub-domains of query optimization, notably parametric query optimization [12, 16, 17] and multi-objective parametric query optimization [33], where it is common to approximate the cost of query plans via piecewise-linear functions. The purpose here is however to model the dependency of plan cost on unknown parameters while traditional approaches such as dynamic programming are used to find the optimal join order. None of the aforementioned publications transforms the join ordering problem into a MILP and the same applies for additional related work that we discuss in Section 2.

A MILP is specified by a set of variables with either continuous or integer value domain, a set of linear constraints on those variables, and a linear objective function that needs to be minimized. An optimal solution to a MILP is an assignment from variables to values that minimizes the objective function. We sketch out next how we transform the join ordering problem into a MILP.

Left-deep query plans can be represented as follows (we simplify by not considering alternative operator implementations while the extensions are discussed later). For a given query, we can derive the total number of required join operations from the number of query tables. As we know the number of required joins in advance, we introduce for each join operand and for each query table a binary variable indicating whether the table is part of that join operand. We add linear constraints enforcing for instance that single tables are selected for the inner join operands (a particularity of left-deep query plans), that the outer join operands are the result of the prior join (except for the first join), or that join operands have no overlap. The result is a MILP where each solution represents a valid left-deep query plan.

This is not yet useful: we must associate query plans with cost in order to obtain the optimal plan from the MILP solver. The cost of a query plan depends on the cardinality (or byte size) of intermediate results. The cardinality of an intermediate result depends on the selected tables and on the evaluated predicates. We introduce a binary variable for each predicate and each intermediate result, indicating whether the predicate has been evaluated to reduce cardinality. Predicate variables are restricted by linear constraints that make it impossible to evaluate a predicate as long as not all query tables it refers to are present in the corresponding result. The cardinality of the join of a set of tables on which predicates have been evaluated is usually estimated by the product of table cardinalities and predicate selectivities. As we cannot directly represent a product via linear constraints, we focus on the logarithm of the cardinality: the logarithm of a product is the sum of the logarithms of the factors. Based on our binary variables representing selected tables and evaluated predicates, we calculate the logarithm of the cardinality for all intermediate results that appear in a query plan. Based on the logarithm of the cardinality, we approximate the cost of query plans via sets of linear constraints and via auxiliary variables.

We must approximate cost functions since the cost of standard operators is usually not linear in the logarithm of input and output cardinalities. We can however choose the approximation precision by choosing the number of constraints and auxiliary variables. This allows in principle arbitrary degrees of precision. Also note that there are entire subdomains of query optimization in which it is standard to approximate plan cost functions via linear functions [12, 16, 17, 33]. Approximating plan cost via linear function is therefore a widely-used approach.

Our goal here was to give a first intuition for how our transformation works and we have therefore considered join order alone and in a simplified setting. Later we show how to extend our approach for representing alternative operator implementations, complex cost models taking into account interesting orders and the evaluation cost of expensive predicates, or richer query languages.

We formally analyze our transformation in terms of the resulting number of constraints and variables. We integrated a prototypical implementation of our approach into the Postgres database system. In our experimental evaluation, we compare our approach against the original Postgres optimizer and several variants. Our results are encouraging: the MILP approach generates guaranteed near-optimal query plans for queries joining up to ca. 40 tables within less than one minute of optimization time in average. Such query sizes are far beyond the reach of traditional dynamic programming based optimization algorithms.

The original scientific contributions of this paper are the following:

- We show how to reformulate query optimization as MILP problem.
- We analyze the problem mapping and express the number of variables and constraints as function of the query dimensions.
- We evaluate our approach experimentally and compare against a classical dynamic programming based query optimizer.

The remainder of this paper is organized as follows. We discuss related work in Section 2. In Section 3, we introduce our formal problem model. Section 4 describes how we transform join ordering into MILP. Section 5 discusses several extensions. In Section 6, we describe our prototypical implementation in the Postgres database system and experimentally compare our approach against the original Postgres optimizer and several variants. In Appendix A, we formally analyze the size of the MILP problems generated by our approach. Appendix B contains additional experimental results providing more insights on the performance of the MILP optimizer. We finally present an experimental study of the impact of MILP optimization on query execution times in Appendix C.

2. RELATED WORK

MILP representations have been proposed for many optimization problems in the database domain, including but not limited to multiple query optimization [11], index selection [26], materialized view design [37], selection of data samples [1], or partitioning of data for parallel processing [2]. In the areas of parametric query optimization and multi-objective parametric query optimization it is common to model the cost of query plans by linear functions that depend on unknown parameters [12, 16, 17, 33]. None of those prior publications formalizes however the join ordering and operator selection problem as MILP.

Query optimization algorithms can be roughly classified into exhaustive algorithms that formally guarantee to find optimal query plans and into heuristic algorithms which do not possess those formal guarantees. Exhaustive query optimization algorithms are often based on dynamic programming [27, 35, 22, 23]. We compare against such an approach in our experimental evaluation.

Our MILP-based approach to query optimization can be used as an exhaustive query optimization algorithm since we can configure the MILP solver to return a guaranteed-optimal solution. The MILP solver can also be configured to return solutions that are guaranteed near-optimal (i.e., the cost of the result plan is within a certain factor of the optimum) or to return the best possible plan within a given amount of time. This makes the MILP approach more flexible than typical exhaustive query optimization algorithms. Furthermore, MILP solvers posses the anytime property, meaning that they produce multiple plans of decreasing cost during optimization. The development of anytime algorithms for query optimization has recently been identified as a research challenge [8]. Transforming query optimization into MILP immediately yields anytime query optimization. Note that anytime algorithms for multi-objective query optimization [32] cannot speed up traditional query optimization with one plan cost metric.

The parallelization of exhaustive query optimization algorithms (not to be confused with query optimization for parallel execution) is currently an active research topic [13, 14, 28, 36]. MILP solvers such as Cplex or Gurobi are able to exploit parallelism and transforming query optimization into MILP hence yields parallel query optimization as well. The development of parallel query optimizers for new database systems requires generally significant investments [28]; the amount of code to be written can be significantly reduced by using a generic solver as optimizer core.

Various heuristic and randomized algorithms have been proposed for query optimization [3, 6, 18, 29, 31, 30, 34]. In contrast to many exhaustive algorithms, most of them possess the anytime property and generate plans of improving quality as optimization progresses. Those approaches can however not give any formal guarantees at any point in time about how far the current solution is from the optimum. MILP solvers provide upper-bounds during optimization on the cost difference between the cost of the current solution and the theoretical optimum. Such bounds could be used to stop optimization once the distance reaches a threshold. Randomized algorithms do not offer that possibility and the returned solutions may be arbitrarily far from the optimum.

3. MODEL AND ASSUMPTIONS

The goal of query optimization is to find an optimal or near-optimal plan for a given query. It is common to introduce new query optimization algorithms by means of simplified problem models. We also use a simple query and query plan model throughout most of the paper while we discuss extensions to richer query languages and plan models as well.

In our simplified model, we represent a query as a set Q of tables that need to be joined together with a set P of binary predicates that connect the tables in Q (extensions to nested queries, queries with aggregates, queries with projections, and queries with non-binary predicates will be discussed). For each binary predicate $p \in P$, we designate by $T_1(p), T_2(p) \in Q$ the two tables that the predicate refers to. Predicates can only be evaluated in relations in which both tables they refer to have been joined.

We assume in the simplified problem model that one scan and one binary join operator are available. As we consider binary joins, a query with n tables requires $n-1$ join operations. A query plan is defined by the operands of those $n-1$ join operations, more precisely by the tables that are present in those operands. We consider left-deep plans. For left-deep query plans, the inner operand is always a single table; the outer operand is the result from the previous join (except for the outer operand of the first join which is a single table).

Query plans are compared according to their execution cost. The execution cost of a plan depends on the cardinality of the intermediate results it produces. We write $Card(t) \geq 1$ to designate the cardinality of table t and $Sel(p) \in (0, 1]$ to designate the selectivity of predicate p. We assume in the simplified model that the cardinality of the join between several tables, after having evaluated a set of join predicates, corresponds to the product of the table cardinalities and the predicate selectivities. We hence assume in the simplified model uncorrelated predicates while extensions to correlated predicates will be discussed. We generally assume that the execution cost of a query plan is the sum of the execution cost of all its operations.

We translate the problem of finding a cost-minimal plan for a given query into a mixed integer linear programming problem (MILP). A MILP problem is defined by a set of variables (that can have either integer or continuous value domains), a set of linear constraints on those variables, and a linear objective function on those variables that needs to be minimized. A solution to a MILP is an assignment from variables to values from the respective domain such that all constraints are satisfied. An optimal solution minimizes the objective function value among all solutions.

4. JOIN ORDERING APPROACH

The join ordering problem is usually solved by algorithms that are specialized for that problem and run inside the query optimizer. We adopt a radically different approach: we translate the join ordering problem into a MILP problem that we solve by a generic MILP solver.

MILP is an extremely popular formalism that is used to solve a variety of problems inside and outside the database community. By mapping the join ordering problem into a MILP formulation, we benefit from decades of theoretical research in the area of MILP as well as from solver implementations that have reached a high level of maturity. By linking query optimization to MILP, we make sure that query optimization will from now on indirectly benefit from all theoretical advances and refined implementations that become available in the MILP domain.

We explain in the following our mapping from a join ordering problem to a MILP. We describe the variables and constraints by which we represent valid join orders in Section 4.1. We show how to model the cardinality of join operands in Section 4.2. In Section 4.3 we associate plans with cost values based on the operand cardinalities.

Note that we introduce our mapping by means of a basic problem model in this section while we discuss extensions to the query language, plan space, and cost model in Section 5.

4.1 Join Order

A MILP program is characterized by variables with associated value domains, a set of linear constraints on those variables, and a linear objective function on those variables that needs to be minimized. Table 1 summarizes the variables that we require to model join ordering as MILP problem and Table 2 summarizes the associated constraints. We introduce them step-by-step in the following.

We start by discussing the variables and constraints that we need in order to represent valid left-deep query plans. Later we discuss the variables and constraints that are required to estimate the cost of query plans.

We represent left-deep query plans for a query Q as follows. For the moment, we assume that only one join operator and one scan operator are available while we discuss extensions in Section 5. Under those assumptions, a query plan is specified by the join operands. We introduce a set of binary variables tio_{tj} (short for *Table In Outer join operand*) with the semantic that tio_{tj} is one if and only if query table $t \in Q$ appears in the outer join operand of the j-th join. Note that we use the term outer operand as synonym for left join operand (and analogue for the term inner operand). We numerate joins from 0 to j_{max} where j_{max} is determined by the number of query tables. Analogue to that, we introduce a set of binary variables tii_{tj} (short for *Table In Inner join operand*) indicating whether the corresponding table is in the inner operand of the j-th join.

The variables representing left-deep plans have binary value domains. Note that not all possible value combinations represent a valid left-deep plan. For instance, we could represent joins with empty join operands. Or we could build plans that join only a subset of the query tables and are therefore incomplete. We must impose constraints in order to restrict the considered value combinations to the ones representing valid and complete left-deep plans.

Left-deep plans are characterized by the particularity that the inner operand consists of only one table for each join. We capture that fact by the constraint $\sum_t tii_{tj} = 1$ which we need to introduce for each join j. A similar constraint restricts the table selections for the outer operand of the first join (join index $j = 0$) as only one table can be selected as initial operand. For the following joins (join index $j \geq 1$), the outer join operand is always the result of the previous join which is another characteristic of left-deep plans. This translates into the constraints $tio_{tj} = tii_{t,j-1} + tio_{t,j-1}$.

The latter constraint actually excludes the possibility that the same table appears in both operands of a join (since the result of the sum between $tii_{t,j-1} + tio_{t,j-1}$ cannot exceed the maximal value of one for tio_{tj}) except for the last join. We add the constraint $tio_{tj_{max}} + tii_{tj_{max}} \leq 1$ for the last join (and optionally for the other joins as well).

The number of joins is one less than the number of query tables. We join two (different) tables in the first join. After that, each join adds one new table to the set of joined tables since the outer operand contains all tables that have been joined so far, since the inner operand consists of one table, and since inner and outer join operands do not overlap. As a result, we can only represent complete query plans that join all tables.

We could have chosen a different representation of query plans with less variables. The problem is that we need to be able to approximate the cost of query plans based on that representation using linear functions. Our representation of query plans might at first seem unnecessarily redundant but it allows to impose the constraints that we discuss next. Also note that MILP solvers typically try to eliminate unnecessary variables and constraints in preprocessing steps. This makes it less important to reduce the number of variables and constraints at the cost of readability.

EXAMPLE 1. *We illustrate the representation of left-deep query plans for the join query $R \bowtie S \bowtie T$. Answering the query requires two join operations. Hence we introduce six variables tio_{tj} for $t \in \{R, S, T\}$ and $j \in \{0, 1\}$ to represent outer join operands and six variables tii_{tj} to represent inner join operands. The join order $(R \bowtie S) \bowtie T$ is for instance represented by setting $tio_{R0} = tii_{S0} = 1$ and $tio_{R1} = tio_{S1} = tii_{T1} = 1$ and setting the other variables representing join operands to zero. This assignment satisfies the two constraints that restrict inner operands to single tables (e.g., $\sum_{t \in \{R,S,T\}} tii_{t1} = 1$ for the second join), it satisfies the constraint restricting the outer operand in the first join to a single table ($\sum_{t \in \{R,S,T\}} tio_{t0} = 1$), and it satisfies the constraints making the outer operand of the second join equal to the union of the operands in the first join (e.g., $tio_{R1} = tio_{R0} + tii_{R0}$).*

4.2 Cardinality

Our goal is to find query plans with minimal cost and hence we must associate query plans with a cost value. The execution cost of a query plan depends heavily on the cardinality of intermediate results. We need to represent the cardinality of join operands and join results in order to calculate the cost of query plans. Inner operands consist always of a single table and calculating their cardinality is straightforward: designating by ci_j (short for *Cardinality of Inner operand*) the cardinality of the inner operand of join num-

Table 1: **Variables for formalizing join ordering for left-deep query plans as integer linear program.**

Symbol	**Domain**	**Semantic**
tio_{tj}/tii_{tj}	$\{0,1\}$	If table t is in outer/inner operand of j-th join
pao_{pj}	$\{0,1\}$	If p-th predicate can be evaluated on outer operand of j-th join
lco_j	$\mathbb{R}$	Logarithm of cardinality of outer operand of j-th join
cto_{rj}	$\{0,1\}$	If cardinality of outer operand of j-th join reaches r-th threshold
co_j/ci_j	$\mathbb{R}_+$	Approximated cardinality of outer/inner operand of j-th join

Table 2: **Constraints for join ordering in left-deep plan spaces.**

Constraint	**Semantic**
$\sum_t tio_{t0}=1/\forall j:\sum_t tii_{tj}=1$	Select one table for outer operand of first join/for all inner operands
$\forall j\forall t: tio_{tj}+tii_{tj}\leq 1$	The tables in the join operands cannot overlap for the same join
$\forall j\geq 1\forall t: tio_{tj}=tii_{t,j-1}+tio_{t,j-1}$	Results of prior join are outer operand for next join
$\forall p\forall j: pao_{pj}\leq tio_{T_1(p)j}; pao_{pj}\leq tio_{T_2(p)j}$	Predicates are applicable if both referenced tables are in outer operand
$\forall j: ci_j=\sum_t Card(t)tii_{tj}$	Determines cardinality of inner operand
$\forall j: lco_j=\sum_t \log(Card(t))tio_{tj}+$ $\sum_p \log(Sel(p))pao_{pj}$	Determines logarithm of outer operand cardinality, taking into account selected tables and applicable predicates
$\forall j\forall r: lco_j - cto_{rj}\cdot\infty\leq\log(\theta_r)$	Activates threshold flag if cardinality reaches threshold
$\forall j: co_j=\sum_r cto_{rj}\delta\theta_r$	Translates activated thresholds into approximate cardinality

ber j, we simply set $ci_j=\sum_t tii_{tj}Card(t)$ where $Card(t)$ is the cardinality of table t.

Calculating cardinality for outer join operands is however non-trivial as we can only use linear constraints: the cardinality of a join result is usually estimated as the product of the cardinalities of the join operands times the selectivity of all predicates that are applied during the join. The product is a non-linear function and does not directly translate into linear constraints.

We circumvent that problem via the following trick. While cardinality is actually defined as the product of table cardinality values and predicate selectivity values, we represent the logarithm of the cardinality instead and the logarithm of a product is the sum of the logarithms of the factors. More formally, given a set $T\subseteq Q$ of query tables such that the set of predicates P is applicable to T (i.e., for each binary predicate in P the two tables it refers to are included in T) and designating by $Card(t)$ for $t\in T$ the cardinality of table t and by $Sel(p)$ the selectivity of predicate $p\in P$, a cardinality estimate is given by $\prod_{t\in T}Card(t)\cdot\prod_{p\in P}Sel(p)$ and the logarithm of the cardinality estimate is $\sum_{t\in T}\log(Card(t))+\sum_{p\in P}\log(Sel(p))$ which is a linear function.

We introduce the set of variables lco_j (short for *Logarithmic Cardinality of Outer operand*) which represents the logarithm of the cardinality of the outer operand of the j-th join. The aforementioned linear formula for calculating the logarithm of the cardinality depends on the selected tables as well as on the applicable predicates. The selected tables are directly given in the variables tio_{tj}. We introduce additional binary variables to represent the applicable predicates: variable pao_{pj} (short for *Predicate Applicable in Outer join operand*) captures whether predicate p is applicable in the outer operand of the j-th join. We currently consider only binary predicates (we discuss extensions later) and, as the inner operands consist of single tables, we do not need to introduce an analogue set of predicate variables for the inner operands.

We denote by $T_1(p)$ and $T_2(p)$ the first and the second table that predicate p refers to. A predicate is applicable to an operand whose table set T contains $T_1(p)$ and $T_2(p)$. We make sure that predicates cannot be applied if one of the two tables is missing by adding for each predicate p and each join j a pair of constraints of the form $pao_{pj}\leq tio_{T_1(p)j}$ and $pao_{pj}\leq tio_{T_2(p)j}$. We currently assume that predicate evaluations do not incur any cost while extensions are discussed later. Under this assumption, applying a predicate has only beneficial effects as it reduces the cardinality of intermediate results and therefore the cost of the following joins. This means that we only need to introduce constraints preventing the solver from using predicates that are inapplicable but we do not need to add constraints forcing the evaluation of predicates explicitly.

Using the variables capturing the applicability of predicates, we can now write the logarithm of the join operand cardinalities. For outer join operands, we set

$$lco_j=\sum_t\log(Card(t))tio_{tj}+\sum_p\log(Sel(p))pao_{pj}$$

and thereby take into account table cardinalities as well as predicate selectivities.

Unfortunately, the cost of most operations within a query plan is not linear in the logarithm of the cardinality values. In the following, we show how to transform the logarithm of the cardinality values into an approximation of the raw cardinality values. This allows to write cost functions that are linear in the cardinality of their input and output. This is sufficient for many but not for all standard operations. Similar techniques to the ones we describe in the following can however be used to represent for instance log-linear cost functions as we describe in more detail in Section 4.3.

We must transform the logarithm of the cardinality into the cardinality itself. This is not a linear transformation and hence we resort to approximation. We assume that a set $\Theta = \{\theta_r\}$ of cardinality threshold values has been defined for integer indices r with $0 \leq r \leq r_{max}$. The set of thresholds can be chosen in order to trade between precision and optimizer performance: the more thresholds we define, the closer the approximation becomes while we need to introduce more auxiliary variables (as discussed next).

We introduce a set of binary variables cto_{rj} (short for *Cardinality Threshold reached by Outer operand*) that indicate for join j and cardinality threshold θ_r whether the cardinality of the outer operand reaches the corresponding threshold value. If threshold θ_r is reached then the corresponding threshold variable cto_{rj} must take value one and otherwise value zero. To guarantee that the previous statement holds, we introduce constraints of the form $lco_j - cto_{rj} \cdot \infty \leq \log(\theta_r)$ for each join j where ∞ is in practice a sufficiently large constant such that the constraint can be satisfied by setting the threshold variable cto_{rj} to one. We do not explicitly enforce that the threshold variable is set to zero in case that the threshold is not reached. The constraints that we introduce next make however sure that the cardinality estimate and therefore the cost estimate increase with every threshold variable that is set to one. Hence, the solver will set the threshold variables to zero wherever it can.

Based on the threshold variables, we approximate cardinality by a step function. We introduce the set of variables co_j representing the raw cardinality of the outer operand of the j-th join and set $co_j = \sum_r cto_{rj}\delta\theta_r$ where the values $\delta\theta_r$ are chosen appropriately such the following holds: if threshold variables cto_{0j} up to cto_{mj} are set to one for join j then the cardinality variable co_j takes a value between θ_m and θ_{m+1} (assuming that thresholds are indexed in ascending order such that $\forall r : \theta_r < \theta_{r+1}$). We can for instance set $\delta\theta_r = \theta_r - \theta_{r-1}$ for $r \geq 1$ and $\delta\theta_0 = \theta_0$.

EXAMPLE 2. *We illustrate how to calculate join operand cardinalities and continue the previous example with join query $R \bowtie S \bowtie T$. We have two joins and introduce therefore four variables (ci_0, ci_1, co_0, and co_1) representing operand cardinalities. Assume that tables R, S, and T have cardinalities 10, 1000, and 100 respectively. We calculate the cardinality of the two inner join operands by summing over the variables indicating the presence of a table in an inner operand, weighted by the cardinality values (e.g., $ci_0 = 10tii_{R0} + 1000tii_{S0} + 100tii_{T0}$). The cardinality of the outer operands can depend on predicates. Assume that one predicate p is defined between tables R and S. We introduce two variables, pao_{p0} and pao_{p1}, indicating whether the predicate can be evaluated in the outer operand of the corresponding join. Predicates can be evaluated if both referenced tables are in the corresponding operand. We introduce four constraints (e.g., $pao_{p0} \leq tio_{R0}$ and $pao_{p0} \leq tio_{S0}$) forcing the value of the predicate variable to zero if at least one of the tables is not present. We introduce two variables storing the logarithm of the outer operand cardinality: lco_0 and lco_1. We assume that the selectivity of p is 0.1. Then the logarithmic cardinality for the first outer join operand is given by $lco_0 = 1tio_{R0} + 3tio_{S0} + 2tio_{T0} - 1pao_{p0}$, assuming that the logarithm base is 10. To simplify the example, we assume that only two cardinality thresholds are considered: $\theta_0 = 10$, and $\theta_1 = 1000$. We introduce four variables cto_{rj} with $r \in \{0, 1\}$ and $j \in \{0, 1\}$ indicating whether the cardinality of the outer join operand reaches each threshold for the first or second join. Each threshold variable is constrained by one constraint (e.g., $lco_0 - \infty \cdot cto_{0,0} \leq 1$). Now we define the cardinality of the outer join operands by constraints such as $co_0 = 10cto_{0,0} + (1000 - 10)cto_{1,0}$. This yields a lower bound on the cardinality. If we know for instance that cardinality values are upper-bounded by 100000 due to the query properties, we can also set $co_0 = 100cto_{0,0} + (10000 - 100)cto_{1,0}$. Then the gap between true and approximate cardinality is at most one order of magnitude.*

4.3 Cost

Now we can for instance sum up the cardinalities over all intermediate results ($\sum_{j \geq 1} cio_j$) and thereby obtain a simple cost metric that is equivalent to the C_{out} cost metric introduced by Cluet and Moerkotte [10]. Join orders minimizing that cost metric were shown to minimize cost according to the cost formulas of some of the standard join operators as well [10]. We will however show in the following how the cost of all standard join operators, namely hash join, sort-merge join, and block nested loop join, can be modeled directly.

The standard cost formula for a hash join operation is linear in the number of pages that the two input operands consume on disk. The number of pages depends on the cardinality of the operands. We have seen in the last subsection how to approximate cardinality based on threshold variables. We can approximate the number of disc pages using the same method: we calculate the number of pages consumed by an operand as a sum over the cardinality related threshold variables. Considering the page count instead of actual cardinality only changes the coefficients in that sum. Note that we simplify by implicitly assuming a constant byte size for each tuple (otherwise, the cardinality alone would not lead to the number of disc pages). We show how to relax that assumption in the next section.

In the worst case, a sort-merge join requires to sort both inputs, followed by the merge phase. We assume in the following that both inputs must be sorted while we generalize in the next section. The cost of a sort-merge join consists of three components in that case (the cost for sorting both inputs and the cost of the merge). The cost of a merge is proportional to the page sizes of both operands. Hence, we can use the same methods as for the hash join to calculate that cost component. The cost of a sort operation is in general not linear in the size of the input. Nevertheless, the same approach as before can be used to approximate arbitrary step functions that are monotone in the logarithm of the cardinality. Hence, we can approximate the cost of a sort operation by summing over the threshold variables approximating cardinality of the corresponding input.

The cost of a block nested loop join is the cost for reading the outer operand and the cost for reading the inner operand repeatedly. The cost for reading the outer operand (if no pipelining is used) is proportional to its size and can be calculated as before. The cost for reading the inner operand is

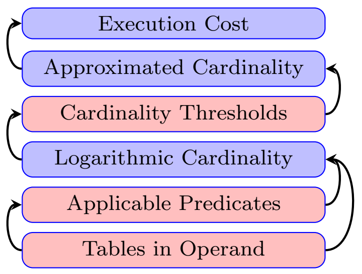


Figure 1: Dependencies between variable groups (binary variables in red, continuous variables in blue) introduced for one intermediate result.

proportional to the product of the size of the inner operand and the number of iterations. The number of iterations depends on the size of the outer operand. Assuming a constant buffer size, we can calculate the number of iterations by a sum over the threshold variables (the ones representing the size of the outer operand) again.

The following formula for calculating the cost of a block nested loop join exploits the fact that the innner operand is always a single table. We denote by $iter_j$ the number of required iterations when performing join number j via a nested loop operator. Also, we denote by $pages(t)$ the number of pages consumed by table t on disc. Now we can express join cost by the formula $\sum_t tii_{tj} \cdot pages(t) \cdot iter_j$. This is a weighted sum over products between a binary variable (the variables tii_{tj} indicating whether table t was selected for the inner operand of join number j) and an integer variable (the variables $iter_j$). This formula is hence not directly linear but the product between a binary variable and a continuous (or integer) variable can be expressed by introducing one auxiliary variable and a set of constraints [4]. The only condition for this transformation is that the continuous variable is non-negative and upper-bounded by a constant. Both is the case (note that we generally only model a bounded cardinality range which implies also an upper bound on the number of loop iterations).

Figure 1 summarizes the variable groups that we introduce for each intermediate result. Join operands are described by binary variables which determine the set of applicable predicates. The logarithm of the join operand cardinality depends on both. We calculate cardinality thresholds based on logarithmic cardinality and approximate cardinality based on those thresholds. This allows us to estimate execution cost for various operators (note in particular that we can approximate non-linear cost functions in the size of input operands).

5. EXTENSIONS

We introduced our mapping for query plans by means of a basic problem model that focuses on join order. We discuss extensions of the query language, of the query plan model, and of the cost model in this section.

Note that not all proposed extensions are necessary in each scenario: the basic model introduced in the last section allows for instance to find join orders which minimize the sum of intermediate result sizes. Such join orders are optimal according to many standard operator cost functions [10]. It is therefore in many scenarios possible to obtain good query plans based on the join order that was calculated using the basic model. To transform a join order into a query plan, we choose optimal operator implementations based on the cardinality of the join operands, we evaluate predicates as early as possible (predicate push-down), and we project out columns as soon as they are not required anymore.

An alternative is to let the MILP solver make some of the decisions related to projection, predicate evaluation, and join operator selection. We show how this can be accomplished if desired. In addition, we discuss extensions of the cost and query model.

5.1 Predicate Extensions

We show how to deal with n-ary, correlated, and expensive predicates. So far we have considered binary predicates. If a predicate p depends on n tables $T_1(p)$ to $T_n(p)$ then we simply introduce constraints of the form $pao_{pj} \leq T_i(p)$ for $1 \leq i \leq n$. This forces variables pao_{pj} to zero if at least one required table is not present.

We assume so far that predicates are uncorrelated. Then the selectivity of a conjunction of predicates is the product of the selectivity values of its components. Predicate independence is a common assumption but not always realistic. Assume a correlated group P_{cor} of predicates such that the selectivity of their conjunction differs significantly from their selectivity product. We introduce a new predicate g that represents the correlated predicate group. The selectivity $Sel(g)$ is chosen in a way such that $Sel(g) \cdot \prod_{p \in P_{cor}} Sel(p)$ yields the correct selectivity, taking correlations into account. So the selectivity of g *corrects* the erroneous selectivity based on simplifying assumption. We introduce a new variable pao_{gj} for g and integrate it into the formulas for logarithmic cardinality as if it were just another predicate. We make sure that g is only activated if all correlated predicates have been evaluated by requiring $pao_{gj} \geq 1 - |P_{cor}| + \sum_{p \in P_{cor}} pao_{pj}$ and $pao_{gj} \leq pao_{pj}$ for each $p \in P_{cor}$.

Example 3. *We extend our running example from Section 4 by adding a second predicate q. The previous predicate p connects tables R and S while q connects S and T. Predicate p still has selectivity* 0.1 *while predicate q has selectivity 0.01. However, both predicates are correlated such that the combined predicate $p \wedge q$ has only selectivity* 0.005 *(instead of selectivity* $0.01 \cdot 0.1 = 0.001$*). We model correlation by introducing binary variables of the form pao_{gj} for each intermediate result j. We refine the formula for calculating the logarithm of the cardinality of intermediate results by setting $lco_j = \log(10) tio_{Rj} + \log(1000) tio_{Sj} + \log(100) tio_{Tj} + \log(0.1) pao_{pj} + \log(0.01) pao_{qj} + \log(5) pao_{gj}$. We add constraints of the form $pao_{gj} \leq pao_{pj}$ and $pao_{gj} \leq pao_{qj}$ to ensure that the correlation factor is only considered if both predicates are evaluated. Finally, we add constraints of the form $pao_{gj} \geq pao_{pj} + pao_{qj} - 1$ to ensure that correlation is considered if both correlated predicates are evaluated.*

So far we have assumed that predicate evaluation is not associated with cost. We constrained the variables pao_{pj} only to zero if required tables are not in the operand. We did not explicitly force them to one at any point since, as they reduce cardinality, their evaluation reduces cost and the MILP solver will generally choose to evaluate them as early as possible.

This model is not always appropriate. If predicate evaluations are expensive then it can be preferable to postpone their evaluation [9, 15, 20]. The predicate-related variables pao_{pj} influence the cardinality estimates of join operands. They capture whether the corresponding predicate *was* already evaluated as otherwise it cannot influence cardinality. We cannot use those variables directly to incorporate the cost of predicate evaluations. The effect on cardinality of having evaluated a predicate once will persist for all future operations. The evaluation cost needs however only to be payed once. We introduce additional variables pco_{pj} (short for *Predicate evaluation Cost for Outer operand*) and set $pco_{pj} = pao_{p,j+1} - pao_{p,j}$. Intuitively, the predicate was evaluated in the current join if it is evaluated in the input to the next join but not in the input of the current join. The sum $\sum_j pco_{pj}co_j$ yields the evaluation cost associated with predicate p (we can additionally weight by a factor that represents predicate evaluation cost per tuple). This is not a linear function as we multiply variables. We have however a product between a binary variable and a continuous variable again. As before, we can transform such expressions into a set of linear constraints and a new auxiliary variable [4].

Now that evaluation of predicates is not automatically desirable anymore, we must introduce additional constraints making sure that all predicates are evaluated at the end of query execution. Designating by j_{max} the index of the last join, we simply set $pao_{p,j_{max}+1} = 1$. This means that each predicate that was not evaluated before the last join must be evaluated during the last join. We finally introduce constraints making sure that no predicate is initially evaluated.

5.2 Projection

Our cost formulas have so far been based on cardinality alone as we have assumed a constant byte size per tuple. This is of course a simplification and we must in general take into account the columns that we project on and their byte sizes. We designate by L the set of columns over all query tables. By $Byte(l)$ we denote the number of bytes per tuple that column $l \in L$ requires. We introduce one variable clo_{jl} (short for *CoLumn in Outer operand*) for each join j and each column $l \in L$ to indicate whether column l is present in the outer operand of join j (and analogue variables for the inner operands). Then a refined formula for the estimated number of bytes consumed by the outer operand is $co_j \cdot \sum_{l \in L} clo_{jl}Byte(l)$. This is the sum over products between a constant ($Byte(l)$), a binary variable (clo_{jl}), and a continuous variable that takes only non-negative values (co_j). This formula can be expressed using only linear constraints using the same transformations that we used already before [4]. For the inner operand, we can estimate the byte size by summing over the column variables, weighted by the column byte size as well as by the cardinality of the table that the column belongs to.

We must still constrain the variables clo_{jl} to make sure that only valid query plans can be represented. First of all we must connect columns to their respective tables. If the table associated with a column is not present then the column cannot be present either in a given operand. If column l is associated with table t then the constraint $clo_{jl} \leq tio_{tj}$ forces the column variable to zero if the associated table is not present. Not selecting any columns would be the most convenient way for the optimizer to reduce plan costs. To prevent this from happening, we must enforce that all columns that the query refers to are in the final result. Also, we must enforce that all columns that predicates refer to are present once they are evaluated. We introduced variables indicating the immediate evaluation of a predicate during a specific join. Those are the variables that need to be connected to the columns they require via corresponding constraints. We must also make sure that a column cannot reappear in later joins after it has been projected out (otherwise that would be a convenient way of reducing intermediate result sizes while still satisfying the constraints requiring certain columns in the final result). Introducing constraints of the form $clo_{jl} \geq clo_{j+1,l}$ satisfies that requirement.

5.3 Choosing Operator Implementations

We have already discussed the cost functions of different join operator implementations in the last section. So far we have however assumed that only one of those cost functions is used to calculate the cost for all joins. This allows to select optimal operator implementations after a good join order, minimizing intermediate result sizes, has been found. We can however also task the MILP solver to pick operator implementations as we outline in the following.

Denote by I the set of join operator implementations. We have shown how to calculate join cost for each of the standard join operators. We can introduce a variable pjc_{ji} (short for *Potential Join Cost*) for each join j and for each operator implementation $i \in I$ representing the cost of the join if that operator is used. We use the term *potential* since whether that cost is actually counted depends on whether or not the corresponding operator implementation is selected.

We introduce binary variables jos_{ji} (short for *Join Operator Selected*) to indicate for each operator implementation i and join j whether the operator was used to realize the join. We require that exactly one implementation is selected for each join as expressed by the constraint $\sum_i jos_{ji} = 1$. Based on potential cost and operator selections, we introduce for each operator the actual join cost ajc_{ji}. The actual join cost is *zero* for a specific operator if that operator is not selected. Otherwise, the actual join cost equals the potential join cost. We have the following relationship between potential and actual join cost $ajc_{ji} = jos_{ji} \cdot pjc_{ji}$. Here we multiply a binary with a non-negative continuous variable and can apply the same linearization as before [4]. The sum over the actual join cost variables over all operator implementations yields the cost of each join operation.

Example 4. *We extend our running example by considering two join operators: a hash join (H) and a sort-merge join (S). To model the choice between those two join operators, we introduce binary variables of the form jos_{jH} and jos_{jS} for each join $j \in \{0,1\}$. Exactly one operator must be selected for each join. We add constraints of the form $jos_{jH} + jos_{jS} = 1$ to express that fact. We also introduce for each join two continuous variables pjc_{jH} and pjc_{jS}, each calculating the jost of join j if the corresponding operator implementation is used (that cost can be calculated as described in Section 4.3). Finally, we introduce two continuous, auxiliary variables ajc_{jH} and ajc_{jS} for each join. We constrain those variables by setting $ajc_{jH} = jos_{jH} \cdot pjc_{jH}$ (this formula can be linearized by introducing additional variables, we omit this transformation due to space restrictions). We* *add join cost over all potential operator implementations to obtain the cost of join j:* $ajc_{jH} + ajc_{jS}$.

5.4 Intermediate Result Properties

Alternative join operator implementations can sometimes produce intermediate results with different physical properties. Tuple orderings are perhaps the most famous example [27]. If tuples are produced in an interesting order then the cost of successive operations can be reduced. Also, whether an intermediate result is written to disc or remains in main memory is a physical property that influences the cost of successive operations.

Assume that we consider a set X of relevant intermediate result properties. Then we can introduce a binary variable ohp_{jx} (short for *Outer operand Has Property*) indicating whether the outer operand of the j-th join has property x. We can model the effect of intermediate result properties in two ways. We can either introduce new join operators that only become applicable if the operands have certain properties or we add new terms to the cost functions of join operators that account for operand properties.

If we introduce new join operators (e.g., a pipelined version of the block nested loop join that becomes only applicable if the outer operand remains in memory) then we need to constrain their applicability. If jos_{ji} represents the choice of a new operator implementation i for a specific join j then we introduce constraints of the form $jos_{ji} \leq UB$. Expression UB is one if all required result properties are present in the join operand and zero or smaller otherwise. Sometimes, it is more convenient to integrate the effect of physical result properties into the cost formulas of the operators. In that case, we need to subtract terms representing cost savings from the formulas for the potential join cost (see Section 5.3). We must also constrain the variables representing result properties. For each variable ohp_{jx}, we introduce an upper bound that may depend on the selected operator for join $j - 1$ and on the properties of the operands for that join. The bound is one if the combination of join operator and input guarantees property x and at most zero otherwise.

EXAMPLE 5. *We extend our running example by considering interesting orders. Assume that the two aforementioned predicates p and q are equality predicates on column a of table R. The cost of a sort-merge join with either p or q as join condition decreases if one or both join operands are ordered by a. We introduce a binary variable* ohp_{1a} *indicating whether the result of the first join (and at the same time the outer operand of the second join) is ordered by a or not (we do not introduce a corresponding variable for the second join as the physical properties of the result cannot speed up any future operations). We assume no specific tuple order in the base tables. The result of the first join is ordered if a sort-merge join is used for the first join and if the join condition is either p or q. We add the constraints* $ohp_{1a} \leq jos_{0S}$ *and* $ohp_{1a} \leq pao_{1p} + pao_{1q}$ *to express that fact. If the outer operand of the first join is already ordered then we save the cost of one sort operation. Denote by* C_S *the cost of sorting the result of the first join (*C_S *is a continuous variable that depends on the cardinality of the operand). We adapt the cost of the second join by subtracting the term* $C_S \cdot ohp_{1a}$ *from the formula for* pjc_{1S} *that was described in the example for Section 5.3. The resulting formula can be linearized [4].*

Component	Language	LOC
Postgres optimizer modification	C	32
Transformation to MILP	Java	281
Greedy optimizer	Java	41
Glue & Benchmarking	Java	74

Table 3: Breakdown of added lines of code for integer programming based query optimization.

5.5 Extended Query Languages

We have already implicitly discussed several extensions to the query language in this section. We discussed how non-binary predicates and projection are supported. This gives us a system handling select-project-join (SPJ) queries.

It is common to introduce query optimization algorithms using SPJ queries for illustration. There are standard techniques by which an optimization algorithm treating SPJ queries can be extended into an algorithm handling richer query languages.

The seminal paper by Selinger [27] describes how complex SQL statements with nested queries can be decomposed into several simple query blocks that use only selection, projection, and joins; the join order optimization algorithm is applied to each query block separately. Later, the problem of unnesting a complex SQL statement containing aggregates and sub-queries into simple SPJ blocks has been treated as a research problem on its own; corresponding publications focus on the unnesting algorithms and use join order optimization algorithms as a sub-function (e.g., [24]).

6. EXPERIMENTAL EVALUATION

We experimentally evaluated a prototypical implementation of a MILP based query optimizer inside the Postgres database system. Section 6.1 describes the prototypical implementation of our approach inside the Postgres optimizer. Section 6.2 describes and justifies the experimental setup and Section 6.3 the experimental results.

6.1 Implementation

We integrated the approach described in Section 4 into the query optimizer of the Postgres database system. The Postgres optimizer is often used as fundament to demonstrate advanced query optimization techniques as it is open source and highly mature. More precisely, we replaced the function that determines an optimal join order inside the Postgres optimizer. While the original algorithm is based on dynamic programming, we use an integer programming solver to determine an optimal join order instead. Our implementation passes on cardinality and selectivity estimates from the Postgres optimizer to a transformation function that creates the corresponding MILP problem. We use Gurobi 6.5.2 to solve the resulting problem. The transformation function uses the result of a simple greedy algorithm to bound the cardinality of the intermediate results that could be part of the optimal plan. The optimal join order is passed back to the Postgres optimizer which constructs the corresponding query plan.

Table 3 shows a breakdown of the number of code lines added. Our implementation consists of less than 450 code lines in total. This is already a moderate code size for a join ordering approach and, as our code is rather optimized for readability than conciseness, it would have been possible to make it even more succinct. The relatively low implementation overhead demonstrates the advantage of reusing existing optimizer components for join ordering.

Our implementation is prototypical and not yet optimized for performance. We implemented different parts in different languages (in order to leverage existing code) and pass parameters between Postgres and the Java based implementation by writing files to disc. The (already impressive) performance results reported in the next sections correspond therefore to lower bounds on the performance of a more mature optimizer implementation.

6.2 Experimental Setup

Standard benchmarks such as TPC-H or TPC-DS[4] have been created for evaluating database execution engines and not specifically for evaluating query optimizers. This is why queries in those benchmarks are of rather moderate size (measured by the number of joined tables). Hence, we cannot use them to demonstrate the ability of query optimization algorithms to deal with large search spaces.

Instead, we created several databases with many tables based on three popular data sets from Kaggle[5]: a data set containing detailed play-by-play statistics for the 2015 NFL season[6], data about loans issued by the loan lending club in the years 2007 to 2015[7], and results of a large survey among code developers[8]. We stored each of those data sets in a star schema by introducing one dimension table for each column of the original data set. Each dimension table has two columns, the key and the associated value. We fill the value column of a dimension table with the distinct values encountered in the associated column in the original data set. We generate unique identifiers for the key column. The fact table contains as many rows and columns as the original data set. However, values are replaced by a references to the key column of the corresponding dimension tables.

We randomly generated queries joining dimension tables with the fact table. Our queries select all columns, the where clause is a conjunction of equality join predicates linking identifiers in the fact table to the key columns of the corresponding dimension tables. We do not apply any additional filter conditions on the dimension tables. We vary the number of tables in our experiments to show the impact on optimizer performance. The reason that we restrict ourselves to relatively simple queries (e.g., no aggregation, no group-by operations, no nested queries) is the following: our approach replaces only the function in the Postgres optimizer that determines join order. The Postgres optimizer decomposes complex queries into simple SPJ query blocks and strips away grouping and aggregation operations before invoking the join ordering component. As the presence of aggregates and group-bys has therefore no impact on the performance of the join ordering algorithm, we do not experiment with different variants.

We compare our MILP based query optimizer against the original dynamic based Postgres optimizer and several modified versions. A plethora of query optimization algorithms are nowadays available. They can be roughly divided into two categories: algorithms that formally guarantee to produce optimal or near-optimal query plans even in the worst case (such algorithms must have exponential time complexity [7]) and heuristic or randomized algorithms. The latter category of algorithms typically has polynomial time complexity and can easily optimize large queries, the average quality of the generated plans is often satisfactory. However, the execution cost of the generated query plans can be arbitrarily far from the optimum in the worst case. This is why most commercial database systems nowadays implement exponential time query optimization algorithms and use them except for large queries. In order not to compare apples and oranges, we restrict our baselines to query optimization algorithms that belong to the first category (exponential time and worst case guarantees).

We use the original version of the dynamic programming based Postgres optimizer as one of the baselines. Furthermore, we use a modified version of the original optimizer that restricts search to left-deep query plans. Our last baseline is a modified version of the Postgres optimizer that is restricted to left-deep query plans but (unlike the original optimizer) also considers Cartesian product joins (which sometimes lead to optimal query plans). Our MILP based optimizer considers left-deep query plans and also considers plans with Cartesian product joins. We model cardinality with a tolerance of $\sqrt{10} \approx 3.16$ (by choosing cardinality thresholds θ_r as powers of 10 while approximating values between thresholds θ_r and θ_{r+1} by value $\theta_r \cdot \sqrt{10}$, similar to Example 2). We can set optimization time bounds for the ILP solver and will experiment with different values. Note that, unlike randomized algorithms, an ILP solver can provide upper bounds on how far the current solution is from the optimum at most. All of the following experiments are executed on a Windows computer with 12 GB of main memory and a 2.5 GHz Intel i7-6500U CPU.

6.3 Experimental Results

We compare the MILP query optimizer against baselines in terms of optimization time. Figure 2 shows optimization time in milliseconds for different databases as a function of the query size (measured as the number of tables).

We compare six optimization algorithms (the semantics of the green line is discussed later): PG-L is the Postgres optimizer, restricting the plan space to left-deep query plans (and not considering Cartesian product joins as in the original optimizer). PG-LC is the Postgres optimizer considering left-deep query plans and Cartesian product joins. PG is the original Postgres optimizer which considers bushy query plans and no Cartesian product joins. IP-500 is the MILP algorithm with an optimization time budget of at most 500 seconds per query (this setting is suitable for offline optimization of frequently executed queries). IP-50 is the MILP algorithm with an optimization time budget of at most 50 seconds, IP-5 takes at most five seconds of optimization time per query (this setting is suitable for run time optimization). Each data point in Figure 2 and in the following figures, ex-

[4]http://www.tpc.org
[5]https://www.kaggle.com
[6]https://www.kaggle.com/maxhorowitz/nflplaybyplay2015
[7]https://www.kaggle.com/wendykan/lending-club-loan-data
[8]https://www.kaggle.com/freecodecamp/2016-new-coder-survey-

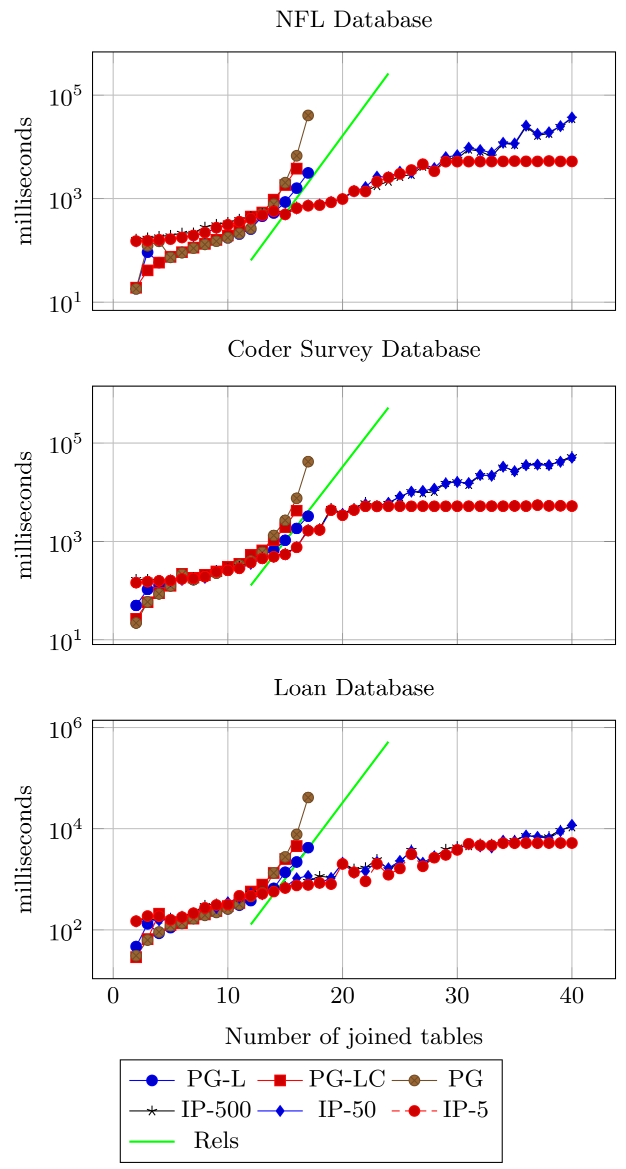


Figure 2: Optimization time until memory out for randomly generated queries on different databases.

cept if noted otherwise, is the median of 15 randomly generated queries.

Optimization time tends to increase for all algorithms and databases as the number of joined tables grows. This is expected as the number of joined tables determines the size of the search space for query optimization. Per default settings, the Postgres database system abandons exhaustive query optimization for queries joining at least 12 tables. We disabled the default setting to evaluate the original and the two modified versions of the Postgres optimizer for larger queries. We are able to optimize queries joining up to 17 tables using the original Postgres optimizer and the version restricted to left-deep query plans. For larger queries, optimization does not finish due to memory outs. For the modified Postgres optimizer that considers Cartesian product joins, we receive memory outs already after 16 tables. This is to be expected as optimizer memory consumption is proportional to the number of potential intermediate results that are considered during dynamic programming: the number of admissible intermediate results increases once we consider Cartesian product joins.

We compare optimization time for the three versions of the Postgres optimizer. The original Postgres optimizer, considering bushy query plans but no Cartesian product joins, needs most optimization time once larger queries are considered. The variant considering only left-deep query plans and no Cartesian product joins is the cheapest in terms of optimization time. Due to memory outs, we are only able to evaluate the dynamic programming based optimizer on a relatively small range of query sizes (which is already far beyond the query size that it treats per default setting). This makes it however difficult to predict optimization time for larger queries. We have therefore inserted another "baseline" (named Rels) that is proportional to the number of intermediate results considered by a dynamic programming algorithm which does not consider Cartesian product joins (calculated according to the formula by Ono and Lohman [25]). This baseline is a lower bound on the asymptotic optimization time growth of all three Postgres optimizer variants (a lower bound only, since optimization time per intermediate result, which is proportional to the number of splits per result into two join operands, will tend to increase as query size grows). We scale that baseline to approximately match optimization time for the fastest Postgres optimizer variant on the largest queries it can treat (where constant time overheads in the optimizer play a less significant role than for small queries). Assuming that large amounts of memory are available, this baseline predicts an optimization time of over eight minutes for the query types we consider as soon as more than 25 tables are joined.

Using integer programming, we are able to optimize queries joining up to 40 tables. This is a query size that is typically considered out of reach for exhaustive query optimization. Using the simple lower bound on optimization time (for left-deep plans and without considering Cartesian product joins) that we established before, optimization time would be 198 days at least for the Postgres optimizer (we would however need over 500 Gigabytes of main memory for optimization, even under the overly optimistic assumption that each intermediate result with associated query plans requires only one byte of storage). Note that our MILP based query optimizer even considers Cartesian product joins which increases the size of the search space significantly (but is sometimes required to find optimal query plans). For optimizing one query joining 40 tables, the MILP optimizer needs less than 54 seconds in average without experiencing any timeouts (for the variant with maximal optimization time budget).

Optimization time seems to grow exponentially for the MILP based query optimizer as well. This is to be expected due to the complexity properties of the query optimization problem. The exponent of the exponential growth seems however to be significantly lower than for traditional dynamic programming based query optimization. For the MILP optimizer versions with stricter timeouts, optimization time converges towards that timeout for large queries. Figure 3 shows the accumulated number of timeouts experienced out of the 45 queries solved per query size. The

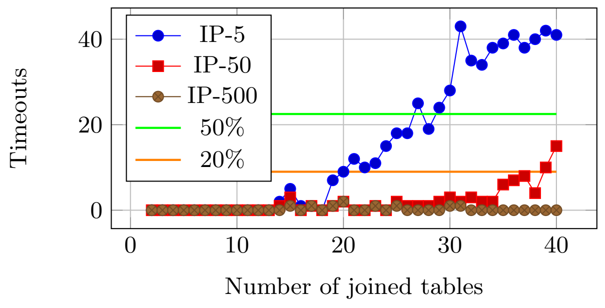


Figure 3: Number of timeouts for MILP optimizers.

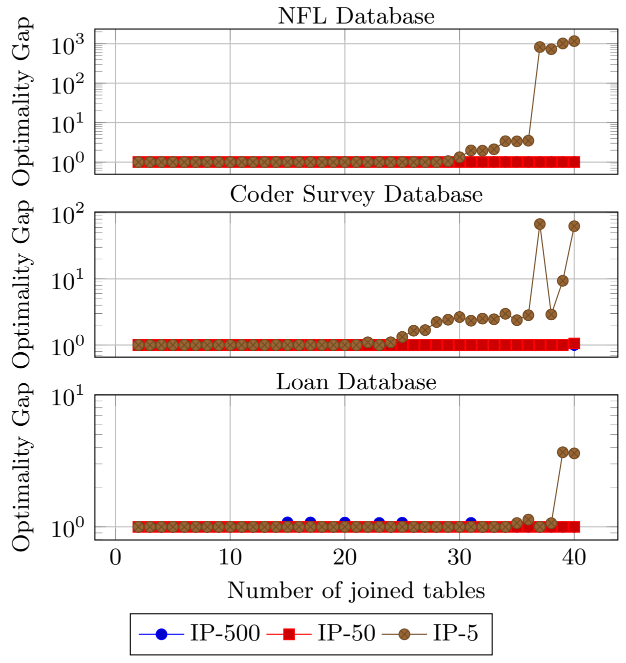


Figure 4: Formally guaranteed upper bound on ratio between returned plan cost and optimal cost.

number of timeouts generally tends to increase in the search space size and in the inverse of the time bound. Note that we describe tendencies that do not hold in each single case (e.g., sometimes the number of timeouts decreases when increasing the query size). This is to be expected as modern integer programming solvers such as Gurobi employ a plethora of heuristics to converge to optimal solutions faster. Optimization time is therefore determined by various factors, only one of them being the search space size (even though probably the most important one). With five seconds of optimization time, the number of timeouts is negligible (i.e., below 20%) for up to 20 tables. For 50 seconds of optimization time, the number of timeouts is negligible for up to 38 tables. As discussed before, average optimization time for 40 tables is only 54 seconds and we experience no timeouts with an optimization time budget of 500 seconds.

Furthermore, a timeout is in many ways less critical for a MILP based query optimizer than for dynamic programming based or even randomized algorithms. After a timeout, a MILP solver typically returns a solution (i.e., a query plan)

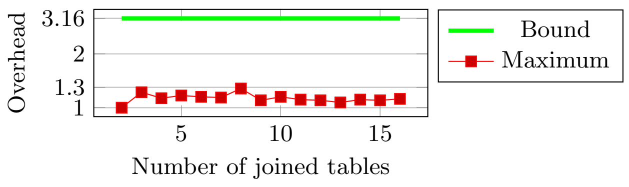


Figure 5: Relative execution cost of plan generated by MILP optimizer compared to Postgres optimizer.

together with a bound limiting the maximal distance between the current solution and a theoretical optimum. This bound can for instance be used to decide whether to dedicate additional optimization time to improve the current solution or whether the gap between current and optimal solution is negligible. We generally observed that a query plan is found quickly in most cases whose cost is guaranteed to be very close to the optimum. Figure 4 shows the bounds on plan optimality (i.e., the factor by which the cost of the current plan is higher than the optimum at most) achieved until the timeout. Significant deviations from the theoretical optimum of one can only be observed for five seconds of optimization time and only for queries of elevated size (up to 36 tables, we have a gap of at most factor 1.15 for the loan data set, 3.5 for NFL data, and 3 for Survey data).

The bounds from the previous figure refer to plan cost in the simplified cost model used by the MILP based query optimizer (i.e., we approximate cardinality via cardinality thresholds). We have finally compared the execution cost of plans generated by the MILP optimizer against plans generated by the Postgres optimizer according to the sophisticated cost model used by Postgres. Figure 5 shows the maximal cost overhead (i.e., the ratio of the execution cost of the MILP plan and the cost of the Postgres optimizer plan) obtained over all test cases for each query size. Note that we only obtain optimal plans for queries with up to 16 tables (from the dynamic programming optimizer considering the same plan space as the MILP optimizer). The results show a moderate worst case overhead of 30% over all 675 relevant test cases, far below the theoretical maximum of around factor 3 (due to cardinality approximation).

7. CONCLUSION

We leverage integer programming solvers for join ordering, a domain where special-purpose algorithms have so far been dominating. Integer linear programming solvers have steadily improved their performance over the past decades and continue to do so at exponential rates. Our experimental evaluation shows that we find guaranteed near-optimal join orders for query sizes that are far beyond the capabilities of classical dynamic programming based query optimization algorithms. As we replace the optimizer core by an existing solver, we obtain a highly configurable query optimizer with low implementation overhead. Our prototypical implementation in Postgres shows that the approach is practical and can be integrated into mature database systems.

8. ACKNOWLEDGMENT

This work was supported by ERC Grant 279804 and by a European Google PhD fellowship.

9. REFERENCES

[1] S. Agarwal, B. Mozafari, and A. Panda. BlinkDB: queries with bounded errors and bounded response times on very large data. In *European Conf. on Computer Systems*, pages 29–42, 2013.

[2] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

[3] K. Bennett, M. Ferris, and Y. Ioannidis. *A genetic algorithm for database query optimization.* 1991.

[4] J. Bisschop. Integer Linear Programming Tricks. In *AIMMS: Optimization Modeling*, page 75ff. 215.

[5] R. E. Bixby. A Brief History of Linear and Mixed-Integer Programming Computation. *Documenta Mathematica*, pages 107–121, 2012.

[6] N. Bruno. Polynomial heuristics for query optimization. In *ICDE*, pages 589–600, 2010.

[7] S. Chatterji and S. Evani. On the complexity of approximate query optimization. In *PODS*, pages 282–292, 2002.

[8] S. Chaudhuri. Query optimizers: time to rethink the contract? In *SIGMOD*, pages 961–968, 2009.

[9] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.

[10] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*, pages 54–67, 1995.

[11] T. Dokeroglu, M. A. Bayir, and A. Cosar. Integer linear programming solution for the multiple query optimization problem. In *Information Sciences and Systems*, pages 51–60. 2014.

[12] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB*, pages 228–238, 1998.

[13] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. In *VLDB*, pages 188–200, 2008.

[14] W.-S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core CPUs. In *SIGMOD*, pages 45–58, 2009.

[15] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. *SIGMOD*, 22(2):267–276, 1993.

[16] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB*, pages 167–178, 2002.

[17] A. Hulgeri and S. Sudarshan. AniPQO: almost non-intrusive parametric query optimization for nonlinear cost functions. In *VLDB*, pages 766–777, 2003.

[18] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. *SIGMOD Record*, 19(2):312–321, 1990.

[19] R. Kaushik, C. Ré, and D. Suciu. General database statistics using entropy maximization. In *Database Programming Languages*, pages 84–99. 2009.

[20] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. *SIGMOD Record*, 23(2):336–347, 1994.

[21] J. A. Lawrence and B. A. Pasternack. *Applied Management Science*. 1997.

[22] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006.

[23] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, pages 9–12, 2008.

[24] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. *VLDB*, pages 91–102, 1992.

[25] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[26] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDEW*, pages 442–449, 2007.

[27] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[28] M. a. Soliman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, R. Baldwin, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, and F. Rahman. Orca: A modular query optimizer architectur for big data. In *SIGMOD*, pages 337–348, 2014.

[29] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDBJ*, 6(3):191–208, 1997.

[30] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. *SIGMOD*, pages 367–376, 1989.

[31] A. Swami and A. Gupta. Optimization of large join queries. In *SIGMOD*, pages 8–17, 1988.

[32] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, pages 1941–1953, 2015.

[33] I. Trummer and C. Koch. Multi-objective parametric query optimization. *VLDB*, 8(3):221–232, 2015.

[34] I. Trummer and C. Koch. Parallelizing query optimization on shared-nothing architectures. In *VLDB*, pages 660–671, 2016.

[35] B. Vance and D. Maier. Rapid bushy join-order optimization with Cartesian products. *SIGMOD*, 25(2):35–46, 1996.

[36] F. M. Waas and J. M. Hellerstein. Parallelizing extensible query optimizers. In *SIGMOD*, pages 871–882, 2009.

[37] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

APPENDIX

A. FORMAL ANALYSIS

State-of-the art MILP solvers use a plethora of heuristics and optimization algorithms which makes it hard to predict the run time for a given MILP instance. It is however a rea-

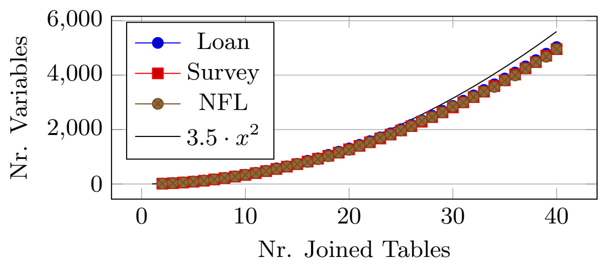


Figure 6: Size of generated integer linear programs as measured by the number of variables.

sonable assumption that optimization time tends to increase in the number of variables and constraints, even if preprocessing steps are sometimes able to eliminate redundant elements. The assumptions that we make here are supported by the experimental results that we present in Section 6: we see a strong (even if not perfect) correlation between the number of variables and constraints and the MILP solver performance.

For the aforementioned reasons, we study in the following how the asymptotic number of variables and constraints in the MILP grows in the dimensions of the query optimization problem from which it was derived. We denote in the following by $n = |Q|$ the number of query tables to join and by $m = |P|$ the number of predicates. By $l = |\Theta|$ we denote the number of thresholds that are used to approximate cardinality values. The following theorems refer to the basic problem model that was presented in Section 4.

THEOREM 1. *The MILP has $O(n \cdot (n+m+l))$ variables.*

PROOF. Give n tables to join, each complete query plan has $O(n)$ joins. We require $O(n)$ binary variables per join to indicate which tables form the join operands, we require $O(m)$ binary variables per operand to indicate which predicates can be evaluated, and we require $O(l)$ continuous variables per operand to calculate cardinality estimates. □

THEOREM 2. *The MILP has $O(n \cdot (n+m+l))$ constraints.*

PROOF. For each join operand we need $O(n)$ constraints to restrict table selections, $O(m)$ constraints to restrict predicate applicability, and $O(l)$ constraints to force the threshold variables to the right value. □

B. MORE ON MILP PERFORMANCE

We show additional experimental results on the performance of the MILP optimizer and the size of the generated MILP programs. Figure 6 shows the size of the linear programs generated for the queries in Section 6. We measure size as the number of variables and report median size as a function of the number of joined tables in Figure 6. We show results for all three scenarios from Section 6. As the queries have the same structure for each scenario, MILP size does not vary significantly accross scenarios.

The variable growth is consistent with the asymptotic results obtained in Section A. We observe close to quadratic growth in the number of query tables (note that the number of predicates is also proportional to the number of tables

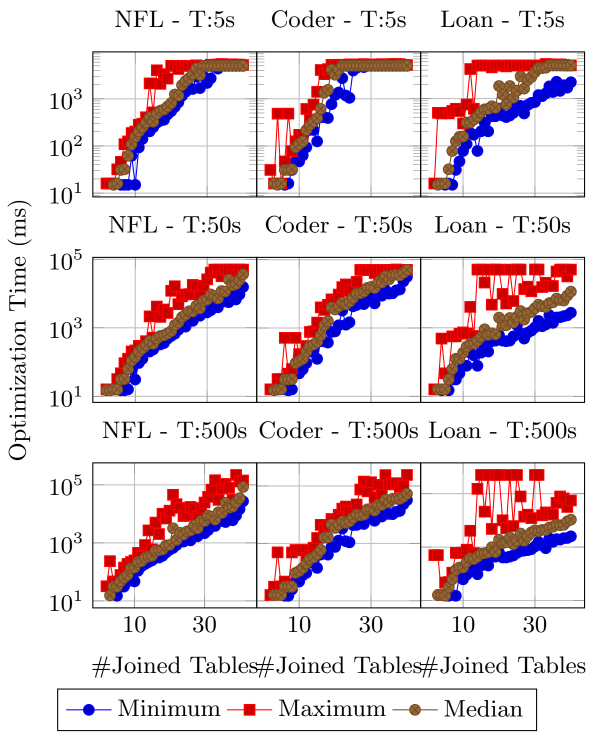
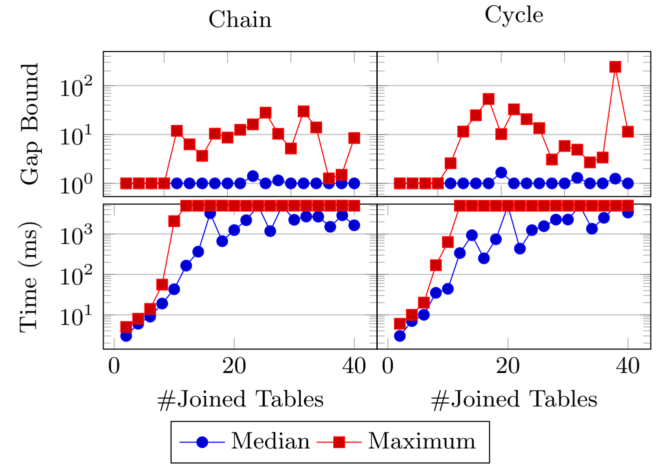


Figure 7: Median, minimal, and maximal MILP optimization time for different scenarios and timeouts.

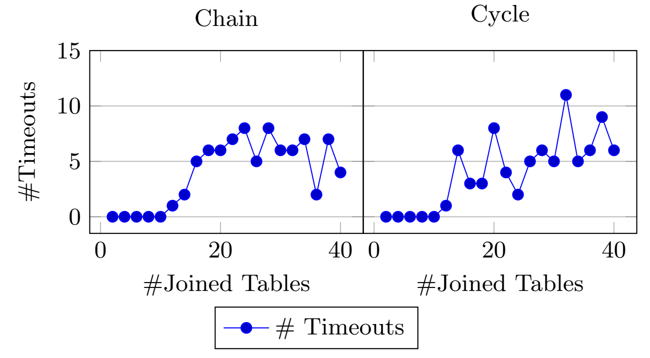
as we consider star queries). A constant factor of approximately 3.5 (see black line) makes the asymptotic development match the experimental results closely.

Section 6 shows median optimization time for the MILP optimizer. Figure 7 expands on that and shows minimal and maximal optimization time for each scenario, timeout bound, and query size. All three (minimum, maximum, and median) tend to increase in the number of joined tables. However, the gap between minimal and maximal optimization time for a fixed number of tables can become quite significant (in particular for the third scenario). This behavior can be explained by the fact that MILP solvers use various heuristics to prioritize their exploration of the search space. While those heuristics work very well in average, they might not work in all cases. Setting a timeout for optimization, as we do in all of our experiments, is therefore important for MILP based query optimization.

So far, we have optimized queries with star-shaped join graphs (which are common in practice). Figure 8 shows results on MILP optimizer performance for different join garph structures. The queries consist of self-joins of a table with two columns (source and target) representing edges in a graph. We filled the table with 3000 randomly generated edges (we choose source and target node with uniform random distribution between 1 and 1000). We generate queries with chain and cycle join graphs. We connect tables via equality predicates, connecting the target column of one table instance to the source column of its successor (i.e., we search for paths in the graph). On each table instance, we restrict with a probability of 0.5 the source column to one specific, randomly selected, value (i.e., we restrict our atten-



(a) Worst case optimality guarantees after optimization (top row) and optimization time (bottom row).



(b) Number of timeouts if optimization time limit is set to five seconds.

Figure 8: MILP optimizer performance for self-join queries with different join graph structures.

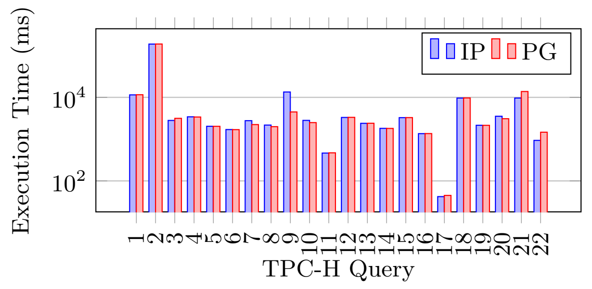


Figure 9: Execution time for TPC-H queries for MILP optimizer and original Postgres optimizer.

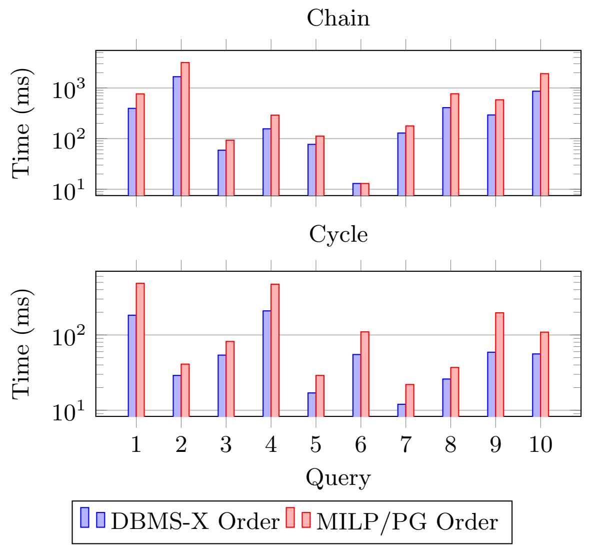


Figure 10: Analyzing overhead when "porting" join order from Postgres to a different database management system.

tion to paths that pass over specific graph nodes). We set a timebound of five seconds on optimization time and report aggregates for 15 randomly generated queries per query size. While a guaranteed optimal solution is often not found starting from around 20 tables, the cost of the generated plans is typically within a factor of at most two around the optimum. There are outliers with a significantly weaker guarantees. However, as MILP optimizers work incrementally, it would be possible to continue optimization (based on conditions that weight the cost of additional optimization against the potential benefit) for those few outliers.

C. IMPACT ON EXECUTION TIME

In Section 6, we evaluate the quality of the plans produced by our MILP optimizer based on the Postgres optimizer cost model. In this section, we verify those results by measuring the real execution times of plans produced by different optimizers. Figure 9 shows execution times for the 22 queries of the TPC-H benchmark in Postgres 9.6 on a TPC-H database with default scaling factor. We use the software and hardware platform that was described in Section 6 for those and the following experiments. We compare the plans generated by the original dynamic programming based Postgres optimizer against the plans generated by the MILP optimizer. Execution times are comparable (median time deviation of less than 0.3%) for all queries except for one (Q9 with an overhead of factor two for the MILP optimizer). While the TPC-H queries are outside of our primary focus (i.e., they are small enough to be handled by dynamic programming based optimizers), we consider those results reasonable.

We have shown that our approach leads to reasonable query plans for Postgres, despite its simplified cost model. Now, we provide some evidence that the approach can work in other database management systems as well. Figure 10 shows the results of porting join orders for a couple of example queries to a commercial database management system that we call DBMS-X in the following. We did not have access to the source code of that database management system but were able to enforce certain join orders by a query reformulation. We compare execution times of query plans in DBMS-X for queries on the graph database that we introduced in Appendix B. We focus on chain and cycle queries joining ten tables (we also generated queries with star-shaped join graph according to the same principles as chain and cycle queries but were not able to execute a single query within several minutes using the original optimizer of DBMS-X). We compare execution times when using the original optimizer of DBMS-X against execution times for the join order generated by the MILP optimizer for Postgres. Our goal is not to beat the original optimizer as the comparison is rather unfair. First, we did not adapt our MILP formulation to the particularities of DBMS-X. Second, as our optimizer is integrated into Postgres, we work with the cardinality estimates of that system which might be less precise than the ones of DBMS-X. Third, we consider only left-deep query plans while the optimizer of DBMS-X considers bushy plans as well. Figure 10 shows that the join orders generated by MILP for Postgres lead to execution plans within factor two of the optimum in most cases and within factor 3 in only three cases. Even without any specific tuning for DBMS-X, the join orders generated by MILP are rather reasonable.

For our final test series, we formulate queries on the popular Musicbrainz database[9]. Musicbrainz is a public music encyclopedia with more than 250,000 users. The data are stored in an SQL database and the corresponding database schema consists of several hundreds of tables. We created several natural queries on that database which require joins between 10 to 30 tables. We compare different query optimizers by measuring optimization time as well as the execution time of the generated query plans.

We created six queries retrieving information on artists and related information such as releases and recordings. To bound execution time, we restrict the number of considered artists to 1000 (Musicbrainz contains information on millions of artists). All queries consist of joins between tables that form semantic units and are connected via key-foreign key constraints in the Musicbrainz database schema. The resulting join graphs are irregular and can neither be classified as cycles, chains, nor star graphs. Our first query joins 10 tables and retrieves information about artists such as their gender, type, and geographic area. For our second query (14 tables), we join artists additionally with information about their releases. For our third query (19 tables), we add more information on those releases such as language, release country, and packaging. For the fourth query (22 tables), we include information on associated recordings. For the fifth query (26 tables), we add more information on recordings such as recording place and associated URLs. For the sixth query (30 tables), we add information on the media (e.g., format and standard identifiers) connected to the releases. We execute the following experiments using the Postgres database since the Musicbrainz scripts for generating the database schema and for loading data are targeted at that database management system.

Figure 11 shows the results of a comparison between three query optimizers. We compare our MILP optimizer (with a timebound on optimization time of five seconds) against the genetic algorithm (Geqo in Figure 11) that Postgres would use by default for the query sizes we consider. The original Postgres optimizer searches a space of join orders but finally uses a heuristic function that may transform a join order into a bushy query plan. To evaluate the effect of that transformation, we also include results for a slightly

[9]https://musicbrainz.org/

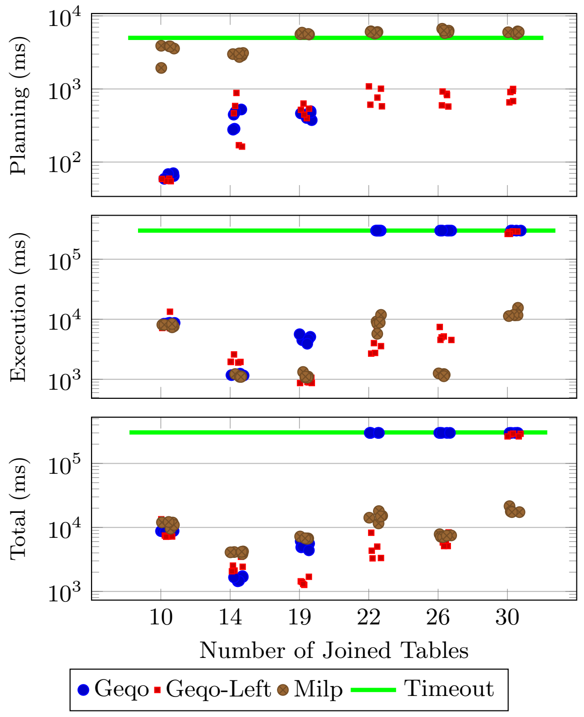


Figure 11: Optimization time, execution time, and total time (optimization and execution) for six queries on the Musicbrainz database.

modified algorithm that omits the final transformation and produces only left-deep query plans (algorithm Geqo-Left in Figure 11).

From the formal perspective, the MILP optimizer is superior to the randomized algorithms as it is the only algorithm capable of providing guaranteed optimal or near-optimal solutions. On the other side, it uses a simplified cost model and considers only left-deep query plans. Figure 11 shows how those properties manifest in optimization and execution time. We show the results for five runs and use jitter in order to separate overlapping data points. We use a timeout of five minutes on execution time for those runs (Postgres does not return planning time in case of a tiemout which is why we do not report optimization time for Geqo-Left and more than 19 tables).

The MILP optimizer uses in general more optimization time than the other two algorithms. On the other side, the execution time of the generated plans is in most cases either minimal or at least close. Considering the sum of optimization time and execution time, investing more time into optimization starts paying off from 26 joined tables (where the overhead in optimization time is approximately matched by reduced execution time). MILP based optimization is clearly the better approach for 30 tables where the two randomized algorithms generate plans taking at least several minutes to execute while the combined execution and optimization time is at most 22 seconds for the MILP optimizer.