ROME: Robust Query Optimization via Parallel Multi-Plan Execution

ZIYUN WEI, Cornell University, USA
IMMANUEL TRUMMER, Cornell University, USA

We present a non-intrusive approach to robust query processing that can be used on top of any SQL execution engine. To reduce the risk of selecting highly sub-optimal query plans, we execute multiple plans in parallel. Query processing finishes once the first of these plans finishes execution.

Plans are selected to be complementary in terms of the intermediate results they generate. This increases robustness to cardinality estimation errors, making cost prediction hard, that concern a subset of candidate results. We present multiple cost-based approaches to selecting plans for robust execution. The first approach uses a simple cost model, based on diversity of intermediate results. The second approach features a probabilistic model, approximating expected execution overheads, given uncertainty on true intermediate result sizes. We present greedy and exhaustive algorithms to select optimal plans according to those cost models. The experiments demonstrate that executing multiple plans in parallel is preferable over executing single plans that are occasionally sub-optimal, as well as over several baselines.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: Query Optimization, Robust Performance, Multiplan Execution, Integer Linear Programming

ACM Reference Format:
Ziyun Wei and Immanuel Trummer. 2024. ROME: Robust Query Optimization via Parallel Multi-Plan Execution. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 170 (June 2024), 25 pages. https://doi.org/10.1145/3654973

1 INTRODUCTION

Computer systems feature an increasing degree of parallelism (due, in part, to fundamental limitations in increasing the clock speed of single CPUs). Database systems typically exploit this parallelism to execute more queries or to process more data in parallel. This strategy is maximally efficient, as long as query plans, chosen for those queries by the optimizer, can be assumed to be near-optimal. However, in practice, query optimization is hard as it relies on accurate cost predictions for alternative plan candidates. Cost estimation may fail for various reasons, including data skew and correlations that make it hard to predict sizes of intermediate results. If so, the query optimizer selects plans that are highly sub-optimal.

We therefore propose a novel strategy to exploit growing degrees of parallelism in current computer systems. Instead of processing more data or executing more queries in parallel, we propose to execute multiple plans for the same query. We can then terminate processing and return a result to the user, once the first plan executed for a given query terminates. Clearly, this strategy creates redundant work in cases where optimal plans are easy to find. On the other hand, if cost

Authors' addresses: Ziyun Wei, zw555@cornell.edu, Cornell University, Ithaca, NY, USA; Immanuel Trummer, itrummer@cornell.edu, Cornell University, Ithaca, NY, USA.

ACM 2836-6573/2024/6-ART170
https://doi.org/10.1145/3654973

estimates are associated with high uncertainty, executing multiple plans works as an insurance. If at least one of the corresponding plans is good, query optimization will terminate quite quickly.

Prior work shows that execution times for a few bad query plans can often dominate execution times for an entire workload [27]. This is due to the fact that plan execution times for the same query often vary by many orders of magnitude. Hence, overheads due to redundant plan executions become quickly negligible, if they help to avoid even a few expensive optimizer mistakes. Beyond that, exploiting large degrees of parallelism for executing single query plans is challenging for many popular database management systems. If parallelization shows diminishing returns for a single plan, executing more plans in parallel can be a better choice. Note that we focus on analytical data processing by single users in this paper. This is a common scenario in which queries are not submitted concurrently, making it impossible to exploit parallelism by executing multiple queries concurrently. If multiple queries are submitted concurrently, it may be preferable to exploit parallelism for executing plans for different queries. However, this is not the scenario we are considering.

The proposed strategy has advantages over other approaches, introduced previously to increase performance robustness in query evaluation. A popular line of research explores the potential of machine learning to make cost estimation more reliable [24, 30, 31]. However, such methods suffer from a cold-start problem in the case of a new database or in case of a dynamic query workload. As shown in our experiments, this makes our approach preferable until large numbers of training samples have been seen. Another popular line of work targets approaches for adaptive query processing [1, 37, 41]. Here, instead of selecting a single query plan, the system switches between different plan candidates (evaluating them based on run time feedback). Adaptive processing typically requires changes to the database management system. Instead, the proposed method can be implemented on top of any SQL engine[1], without changing its code.

More precisely, we obtain alternative plans for the same query by issuing the query repeatedly while changing the settings for the optimizer. This is similar to other recent work, restricting the search for optimal plans by exploiting the traditional query optimizer. While such optimizers are not reliable (motivating this work), they remain useful in pruning an overwhelmingly large space of plan candidates to reasonable alternatives. By optimizing with different optimizer settings, we can obtain a large number of plans for the same query. Executing all of them is prohibitively expensive. This makes it crucial to select a small subset of plans for execution that is maximally efficient.

Starting from a large space of alternative plans, the initial filtering is done by the query optimizer, narrowing the scope to plans that it generates for some optimizer settings. Still, due to the large number of possible optimizer settings, the resulting plan set is still too large. We formulate the problem of selecting an optimal subset of plans for execution as a combinatorial optimization problem. As execution stops once the first parallel plan finishes, a subset of plans is as good as the best plan in that set. This insight motivates specific utility functions, judging the quality of a combination of plans for execution.

We consider multiple problem variants and associated optimization algorithms. First, we consider a relatively simple utility function, counting the number of distinct intermediate results (considering the structure of the query plan alone). Typically, bad plan choices derive from mistakes when estimating the cardinality of intermediate results. Selecting plans that generate similar intermediate results increases the risk of a single estimation error influencing many plans. Together with this utility function, we propose an efficient optimization algorithm that guarantees near-optimal plan sets. In addition, we propose an exhaustive algorithm, based on integer linear programming.

[1]We assume that the SQL engine supports a small set of standard operations for query planning that we discuss in more detail in the following sections.

This algorithm trades search overheads for guarantees optimal solutions. Second, we consider a refined model that associates intermediate results with a probability distribution over possible cardinality values. To initialize those probability distributions, we use the cardinality estimates by the optimizer as a starting point. The quality of a plan set is then determined by the expected value of the minimum execution cost over all selected plans (since execution terminates once the first selected plan terminates). We propose greedy and exhaustive methods to select optimal plan sets, according to these refined metrics.

In the experiments, we compare our approach to multiple baselines on multiple benchmarks, using PostgreSQL as the underlying database management system. We use two benchmarks that are based on real data, making query optimization hard due to skew and correlation. We compare against the original PostgreSQL optimizer (selecting single plans executed with full degree of parallelism), several simple baselines for multiplan selection, the plan bouquets approach [13], adaptive query processing methods, as well as several variants of Bao [30], an approach to configure the PostgreSQL optimizer via configuration parameters, based on reinforcement learning. The experiments demonstrate that executing more plans in parallel, as opposed to executing single plans with maximal degree of parallelism, ultimately pays off in terms of accumulated execution overheads. Also, they show that learning-based approaches suffer from cold-start problems whereas the proposed approach works "out of the box". In summary, the original scientific contributions in this paper are the following:

- We propose a non-intrusive optimization framework, executing complementary query plans in parallel.
- We propose multiple utility models and multiple algorithms for selecting good sets of query plans.
- We analyze the problem of multi-plan selection and the proposed methods formally.
- We compare the proposed method to multiple baselines in experiments, using multiple benchmarks on real data.

The remainder of this paper is organized as follows. Section 2 gives an overview of the ROME architecture and discusses implementation details. Section 3 introduces the formal problem model and associated terminology. Section 4 introduces a simple utility model for selecting plan sets for execution, as well as associated optimization algorithms. Section 5 introduces a more sophisticated utility model, calculating expected execution costs for plan sets, as well as associated optimization methods. Section 6 analyzes the problem as well as the proposed methods formally. Section 7 reports results of experiments while Section 8 discusses prior work.

2 ROME FRAMEWORK

We give an overview of ROME's architecture in Section 2.1 and discuss implementation details in Section 2.2.

2.1 Overview

Figure 1 shows an overview of our approach, executing complementary plans in parallel to reduce the impact of optimizer mistakes. We will refer to this approach as ROME (short for Robust Optimization by Multiplan Execution) in the following. ROME is a lightweight framework on top of relational database management systems. ROME is completely non-intrusive and does not require any changes to the underlying database management system. It is generically applicable and works on any SQL processing engine, provided that the system offers parameters that influence query plan choices. In our experiments, we will use ROME on top of PostgreSQL. However, the approach is not specific to that system.

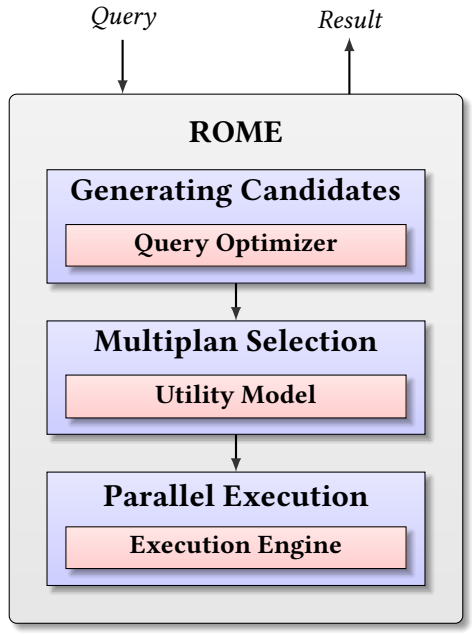

Fig. 1. Overview of ROME.

ROME focuses on SQL queries. It is particularly beneficial in scenarios where query optimization is difficult (e.g., due to skewed data), making it hard for the optimizer to identify near-optimal plans reliably. In those cases, ROME provides insurance against bad optimizer choices by executing multiple complementary plans in parallel. Execution terminates whenever the first of those plans terminates.

As shown in Figure 1, SQL queries are sent to ROME (as opposed to sending them to the underlying database management system directly). Given a query, ROME first generates a set of interesting plan candidates. To do so, ROME optimizes the input query repeatedly but with different optimizer settings. For instance, for Postgres, ROME manipulates flags that determine the set of operator implementations considered by the optimizer. More precisely, it manipulates the same parameters used in other recent work for optimizer steering [30]. By disabling subsets of operator implementations, ROME effectively retrieves local optima for subsets of the query plan space (using the optimizer of the underlying database system to identify those local optima). While those local optima are never better than the global optimum, according to the cost model of the query optimizer, they may turn out to be optimal in the presence of cost and cardinality estimation errors. The approach for selecting plans to execute, described next, is our primary technical contribution in this paper. It can be easily combined with other approaches for generating plan candidates.

ROME executes multiple locally optimal query plans in parallel. However, executing parallel plans is costly and reduces the degree of parallelism that is available for each single plan execution. Therefore, it is prohibitively costly to execute all plan candidates in parallel. Because of that, ROME selects a small set of plans for execution. In the following, we refer to this step as multiplan selection. We treat multiplan selection as a combinatorial optimization problem. Intuitively, our goal is to select a set of plans that is as complementary as possible. This means that, ideally, incorrect assumptions, influencing for instance size estimation for a subset of intermediate results, never

affect all selected plans at the same time. In the following sections, we propose multiple utility models to judge the quality of a particular plan set, as well as associated optimization algorithms.

After selecting plans, ROME executes those plans in parallel (by issuing the same query multiple times via different sessions, varying the optimizer settings in each session to obtain the selected plan). Once the first plan finishes, ROME terminates all parallel executions. This means that execution time is effectively determined by the execution time of the fastest plan in the set. All executed plans generate the same result. Therefore, the result generated by the first plan to finish is returned to the user.

Note that ROME performs redundant work by executing multiple plans for the same query. However, as we will see in the experiments, this approach pays off if it helps to avoid the occasional (but very expensive) optimizer mistake, misleading the optimizer to select plans with an execution cost far above the optimum.

2.2 Implementation Details

The implementation of ROME, evaluated in the experiments, is a Python framework on top of PostgreSQL. Queries, along with access credentials for the associated PostgreSQL database, are submitted to ROME. Upon receiving a new query, ROME generates plan candidates using optimizer flags for PostgreSQL. More precisely, ROME considers optimizer configuration changes of the form `SET [flag] to [value];` where `[flag]` and `[value]` are placeholders. For instance, `enable_nestloop` is one of the flags considered by ROME, and `on` and `off` are the two possible values for this Boolean flag. In this plan generation stage, ROME submits to PostgreSQL SQL scripts that concatenate one or multiple configuration parameter changes (of the aforementioned type), followed by the input query, prefixed by PostgreSQL's `EXPLAIN` command. The `EXPLAIN` command has the effect that the following query is not executed. Instead, PostgreSQL merely returns a description of the execution plan that the optimizer would pick for execution. The selected plan depends on the preceding optimizer configuration changes as they may disable parts of the plan space (by disabling specific operators). ROME tries all combinations of settings for optimizer flags, generating one plan for each such combination. As query planning is very fast, compared to executing queries, the time overhead of this plan generation stage is negligible.

After retrieving alternative plans from PostgreSQL, ROME parses those plans, focusing on the join trees and the associated intermediate results. The approaches for plan selection, described in the following sections, are based on the intermediate results that are generated during plan execution. However, the parsed plans may contain other operators (e.g., aggregates) that do not influence intermediate join results. Given one of the methods described in the following sections, ROME selects a small subset of complementary plan candidates for parallel execution.

For each of the plans selected for parallel execution, ROME creates a sub-process. Each such sub-process submits an SQL script to PostgreSQL. This SQL script contains the optimizer configuration changes that led to the generation of the selected plan, followed by the input query. The only difference to the script used during plan generation is the fact that the `EXPLAIN` keyword is not used as a prefix to the input query. This means that the corresponding query is now executed, using the plan described as output of the `EXPLAIN` command. ROME terminates all sub-processes once the first such sub-process finishes, returning the query result. This query result is then returned to the user.

3 FORMAL MODEL

Our technical contribution in this paper focuses on the multiplan selection problem (see Figure 1). In the following, we introduce this problem and related terminology.

ROME focuses on relational queries. ROME uses parser, optimizer, and execution engine of an underlying database management system. Hence, in principle, ROME supports all queries supported by the underlying SQL engine. Queries are mapped to one or multiple query plans for execution.

Definition 1 (Query Plan). *We model query plans as join trees in the following. For most of this paper, we implicitly assume binary join trees (i.e., the system uses binary join operators). However, the approach is not limited to this scenario and could support non-binary join trees (e.g., if the underlying system uses multiway joins) as well.*

When modeling query plans in the context of multiplan selection, we neglect other operators than joins (e.g., aggregates, grouping, and sorting). However, while other operators are not considered for multiplan selection, they may still appear in the original query plans. Therefore, our implementation supports general SPJGA SQL queries without sub-queries. Query plans are associated with a set of intermediate results that are generated during plan execution.

Definition 2 (Intermediate Result). *We associate inner nodes within join trees representing query plans with intermediate results. For a plan p, we denote by $\mathcal{I}(p)$ (to save space, we also use the notation $\mathcal{I}_p$) the set of intermediate results. Different plans may share intermediate results. If so, an estimation error for one intermediate result may lead to incorrect cost estimates for multiple plans.*

In the following, we will often identify intermediate results by a set of joined tables (e.g., $R \bowtie S \bowtie T$). This notation assumes that all applicable predicates (including unary predicates as well as join predicates) are applied as soon as possible (predicate push-down). Under this assumption, due to the fact that we only compare plans for the same query, intermediate results are equivalent if they join the same tables (e.g., $R \bowtie S \bowtie T$ and $T \bowtie S \bowtie R$ are considered the same intermediate result in the following).

Difficulties in estimating the sizes of those intermediate results are often the reason for sub-optimal plan choices.

Definition 3 (Size Estimates). *During part of the paper, we assume that the underlying database engine features a cost-based query optimizer. To perform cost-based query optimization, systems generally need to estimate the sizes of intermediate results (based on data statistics, typically). ROME obtains size estimates from the underlying optimizer. For an intermediate result i, we denote by $\mathcal{M}(i)$ the size estimate by the optimizer size model.*

The ultimate goal of ROME is to select a set of plans for parallel execution that *minimize the execution time of the fastest plan* (since execution terminates once the first plan does). Of course, if optimizer cost models were reliable, the first plan to finish would always be the one selected by the optimizer by default. However, in practice, cost estimates are associated with uncertainty. Hence, ROME selects a set of complementary plans as insurance against optimizer mistakes. This leads to the following problem.

Definition 4 (Multiplan Selection). *Given a set of plan candidates P, a maximal number n of plans to execute, and a utility function $\mathcal{U} : P \mapsto \mathbb{R}$, mapping plan sets to real-valued utility values, the goal of multiplan selection is to select a set $P^* \subseteq P$ of plans with $|P^*| \leq n$ that maximizes utility among all plan sets with at most n plans.*

ROME generates candidate plans by repeatedly invoking the optimizer of the underlying system with different configuration settings (thereby obtaining locally optimal plans, according to the optimizer, in different plans of the plan space). The following sections introduce multiple utility functions, based on purely structural plan properties or integrating estimates from an optimizer cost model. The goal of the utility function is to serve as proxy for the overarching goal of minimizing execution cost of the fastest executed plan.

4 MAXIMIZING PLAN DIVERSITY

We introduce a first approach to plan selection that is purely based on the plan structure. Section 4.1 discusses a corresponding utility metric for plan sets. Section 4.2 describes how to map multiplan selection with this utility metric to ILP. Section 4.3 analyzes the properties of the problem. This analysis motivates a greedy algorithm, described in Section 4.4 that guarantees near-optimal results.

4.1 Utility Metric

Our goal is to select a set of plans that is robust to cardinality estimation errors. E.g., assume that the actual size of one specific intermediate result deviates significantly from the optimizer estimate. Ideally, this error only affects a subset of plans executed in parallel. If so, the remaining plans (which are optimal within part of the plan space) will finish as expected. In this subsection, we introduce a simple utility metric on plan sets as a proxy for robustness. More precisely, we evaluate the utility of a plan set as the number of distinct intermediate results that appear in those plans.

Definition 5 (Plan Diversity). *Given a set P of alternative plans for the same query, we denote by $\mathcal{U}(P) = |\cup_{p \in P} \mathcal{I}_p|$ the (structural) plan diversity.*

We give an intuitive explanation for why this utility metric is useful. Assume that plans P with $|P| = n$ are executed in parallel. Assume that those plans join $l + 1$ tables (all plans refer to the same query and must join the same number of tables). Denote by u the plan diversity of the plan set.

Lemma 1. *Each unique intermediate result appears in $n \cdot l/u$ plans in average.*

Proof. Plans correspond to binary trees with $l + 1$ leaf nodes (for the joined tables). Hence, the number of inner nodes (representing intermediate results) is l for each plan. The plan diversity u is therefore between l (i.e., all plans share the same intermediate results) and $n \cdot l$ (no intermediate results are shared). Also, dividing $n \cdot l$ intermediate results across u distinct results yields $n \cdot l/u$ occurrences for each intermediate result in average. The same intermediate result can only occur once per plan. Hence, the average number of occurrences is the average number of plans in which the same result appears. □

Hence, by maximizing plan diversity, we minimize the expected number of plans that are affected by one single cardinality estimation error.

4.2 Transformation to ILP

We show how to transform multiplan selection, using the aforementioned utility metric, into integer linear programming. After that transformation, we can apply existing solvers to find an optimal solution.

4.2.1 Decision Variables. We aim to identify a set of plans that cover the maximum number of intermediate results. Each plan is characterized by a specific set of intermediate results. We introduce binary variables m_p to indicate whether a plan $p \in P$ is selected. Similarly, we introduce binary variables, denoted as r_i, to indicate whether an intermediate result i is chosen by any of the plans. Note that we do not need to represent results associated with base tables or the final query result (since both appear in any possible query plan).

4.2.2 Constraints. For each selected plan, all intermediate results covered by that plan must also be selected. This requirement can be expressed through constraints of the form $m_p \leq r_i$ for each plan $p \in P$ and intermediate results $i \in \mathcal{I}_p$. Additionally, in the case of a chosen intermediate result i, it is necessary to include at least one plan that incorporates i. Consequently, we impose the following constraints: $\sum_{p:i \in \mathcal{I}_p} m_p \geq r_i$. Furthermore, the total number of plans cannot exceed the maximally allowed number of concurrent plans, n. Hence, we have the constraint $\sum_{p \in P} m_p \leq n$.

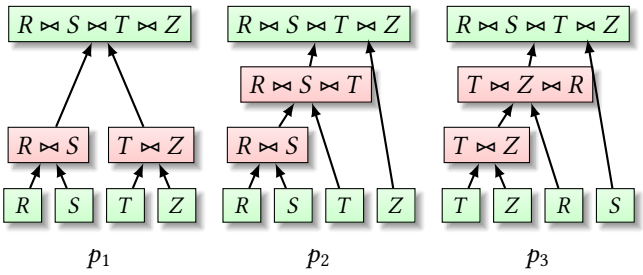

Fig. 2. Example of plan candidates: three query plans p_1, p_2, p_3 joining tables R, S, T, Z. Each inner node represents an intermediate result (results marked in red are relevant when maximizing plan diversity).

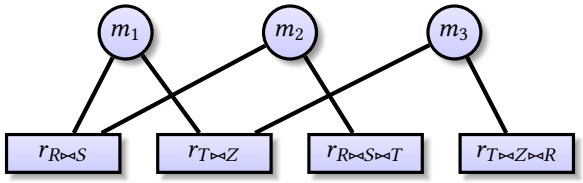

Fig. 3. Decision variables of ILP maximizing plan diversity, together with connecting constraints.

4.2.3 Objective Function. The objective function aims to maximize the sum of selected intermediate results ("plan diversity"), denoted as $(\sum_{i \in \mathcal{I}(P)} r_i) - \epsilon(\sum_{p \in P} m_p)$. In order to prevent the inclusion of a larger subset of plans with an equal number of covered intermediate results, we introduce a penalty term. Here, ϵ represents a sufficiently small value (e.g., 0.001). This conveys our goal of maximizing the number of distinct intermediate results covered by the chosen plans (while avoiding selecting additional plans that do not add new intermediate results).

Example 1. Consider the three query plans represented in Figure 2. They represent alternative plan candidates answering a query joining four tables: R, S, T, Z. To select an optimal plan subset via ILP, we introduce three binary variables, m_1, m_2, m_3, associated with plan candidates, and four binary variables for unique intermediate results (highlighted in red) denoted by $r_{R \bowtie S}$, $r_{T \bowtie Z}$, $r_{R \bowtie S \bowtie T}$, $r_{R \bowtie T \bowtie Z}$. In Figure 3, we visualize those decision variables, along with the structure of the constraints connecting them. Constraints are represented by connecting lines. Note the dependencies between decision variables associated with plans and the variables associated with intermediate results of those plans.

4.3 Problem Analysis

For the following analysis, we consider the utility model motivated in Section 4.1 (plan diversity). We prove several properties of that utility function that inform the design of a greedy algorithm in the following subsection.

Theorem 2. *Plan diversity is non-negative and monotone.*

Proof. Plan diversity is defined as a set cardinality (the cardinality of the set of intermediate results). Hence, the minimal value is zero. Also, adding new plans to the set cannot reduce the number of distinct intermediate results (i.e., it is $|\cup_{p \in P} \mathcal{I}_p| \leq |\cup_{p \in P'} \mathcal{I}_p|$ if $P \subseteq P'$). □

Next, we show that plan diversity has diminishing returns. First, we define this property formally.

Algorithm 1 Select subsets of plan candidates maximizing plan diversity.

```
// Given a set of plan candidates P, select at most n plans maximizing plan diversity for concurrent execution.
function GreedyMPD(P, n)
    // Initialize set of executed plans
    P* ← ∅
    // Repeat adding plans to the optimal set
    while |P*| ≤ min(n, |P|) do
        // Select most complementary plan
        p* ← arg max_{p∈P\P*} U(P* ∪ {p})
        // Did the utility value increase?
        if U(P* ∪ {p}) > U(P*) then
            P* ← P* ∪ {p}
        else
            break
        end if
    end while
    return P*
end function
```

Definition 6. *A set function $f : S \mapsto \mathbb{R}$ is **submodular** if the following holds. If adding an element $s \in S$ to two sets $S_1 \subseteq S_2 \subseteq S$ then $f(S_1 \cup \{s\}) - f(S_1) \geq f(S_2 \cup \{s\}) - f(S_2)$.*

Theorem 3. *Plan diversity is submodular in the set of selected plans.*

Proof. Consider two sets of plans P_1 and P_2. We assume that $P_1 \subseteq P_2$. Assume a new plan p is added to both sets, P_1 and P_2. We denote by I_1 and I_2 the set of intermediate results of p that are not contained in P_1 and P_2 respectively. Thus, $|I_j| = \mathcal{U}(P_j \cup \{p\}) - \mathcal{U}(P_j)(j = 1, 2)$. For any intermediate result $i \in I_2$, it is also in I_1. Otherwise, i only appears in unique intermediate results of P_1, which contradicts $P_1 \subseteq P_2$. Thus, $I_2 \subseteq I_1$, proving that $|I_2| \leq |I_1|$. This proves that plan diversity is submodular. □

4.4 Greedy Algorithm

Consider Algorithm 1. This algorithm selects plans greedily for execution. The input is a set of plans, together with the maximal number of plans that can be executed concurrently. Starting from an empty set, the algorithm iteratively adds the plan that maximizes utility in each iteration (here, utility represents plan diversity). Iterations continue until the maximal number of plans has been selected (or no candidates remain to be selected). Also, the algorithm terminates if no plan increases utility. This termination criterion is reasonable. Due to submodularity, adding a plan to supersets of the currently selected plans cannot yield a higher increase in utility. If no plan achieves an improvement in utility, no plan can increase utility in later iterations either.

Algorithm 1 is seemingly simple. Nevertheless, it guarantees near-optimal results, according to the following theorem. Note that this theorem is based on the properties of the utility function, shown in the previous subsection. By e, we denote Euler's number in the following theorems.

Theorem 4. *Greedy plan selection achieves plan diversity within factor $(1 - 1/e)$ of the optimum.*

Proof. We have shown that plan diversity is non-negative and monotone (Theorem 2) and submodular (Theorem 3). According to Nemhauser [34], greedily adding a bounded number of elements with maximal utility leads to a near-optimal result, within factor $(1 - 1/e)$ of the optimum. Note that terminating early in case of non-increasing utility does not change the utility of the result, as justified previously. □

We will see in the experiments that the greedy algorithm realizes attractive tradeoffs between computational overheads (of plan selection) and result quality.

5 MINIMIZING EXPECTED COSTS

We introduce a more precise model to guide our selection of plans for parallel execution. Section 5.1 explains how we associate intermediate results with a probability distribution, modeling cardinality. Section 5.2 shows how we calculate expected execution costs for plan sets (assuming that execution finishes once the first plan terminates). Section 5.3 introduces a greedy algorithm to find plans with near-optimal execution costs. Finally, Section 5.4 describes a transformation to ILP, enabling us to use existing solvers to find optimal plan sets.

5.1 Modeling Intermediate Result Cardinality

Incorrect cardinality estimates are often at the root of costly query optimizer mistakes. Prior work [27] demonstrates that the impact of erroneous cardinality estimates, due to factors such as data correlation and skew, is significant. In the following, we present a model for selecting plans for parallel execution that takes into account uncertainty in cardinality estimation.

Typically, query optimizers associate intermediate results with point size estimates. However, this neglects uncertainty that arises, e.g., from simplifying assumptions made while estimating the selectivity of query predicates (e.g., assuming that different predicates in the same query are uncorrelated). In addition, estimates are based on data statistics that are typically coarse-grained and fail to take into account dependencies between different columns in the database.

To account for all these types of uncertainty, we model the size of intermediate results as a probability distribution in the following (rather than a point estimate). Intermediate results are created via joins between other intermediate results (or base tables). Given known input sizes l and r (for the left and right join operand), the maximal output size is $l \cdot r$ (i.e., the Cartesian product). Typically, Cartesian product joins are avoided, meaning that join outputs are filtered by a join predicate. In the following, we denote the estimated selectivity of that join predicate by σ. Query optimizers typically assume a point size estimate of $l \cdot r \cdot \sigma$ on the join output size. Instead, we model the join output size as a probability distribution.

We can associate each row j in the Cartesian product with a random variable, o_j, with Bernoulli distribution (it is $o_i = 1$ if the corresponding row satisfies the join predicate). The output size is then given as the sum over those random variables, i.e., $\sum_j o_j$. Assuming independent and identically distributed variables, the output distribution can be modeled by a Binomial distribution. However, the Binomial distribution is inefficient to compute. Instead, we opt to approximate the output distribution by a Poisson distribution. The Poisson distribution generally models the number of successes, given a large number of trials with a fixed success probability. In our case, each trial corresponds to a row in the Cartesian product whereas a success corresponds to a row satisfying the join predicate. The Poisson distribution is known to approximate the Binomial distribution well in cases where the number of trials is large while the success probability is small, e.g., starting from 1,000 trials with a success probability of at most 1% [28]. Given typical table sizes, the selectivity of join predicates as well as the number of Cartesian product rows often satisfy those conditions.

The Poisson distribution is characterized by a single parameter λ, representing the expected value as well as the variance at the same time. To approximate the size of a join result, joining tables with sizes l and r and using a join predicate of selectivity σ, we set $\lambda = l \cdot r \cdot \sigma$ (i.e., we set it to the size estimate used by the query optimizer).

An additional challenge in our context is the following. Different from the situation described previously, we cannot assume that the sizes of join inputs are point estimates. Instead, we must assume that they are associated with uncertainty as well. Even if input sizes follow a Poisson distribution as well, we cannot assume that the output distribution can be modeled as a Poisson distribution in general. Instead of modeling the output distribution analytically, we opt to approximate it by a fixed number of discrete values with associated probabilities, a classical approach in the domain of robust query optimization [9].

For base tables, we assume that cardinality is known (even after applying filter predicates). This reflects the fact that incorrect cardinality estimates are often due to correlations between different predicates [6, 27]. Hence, we model their cardinality as a point estimate (i.e., a special case of a discrete probability distribution that assigns a probability of one to the optimizer estimate). For a given plan, we calculate a probability distribution for each intermediate result bottom-up. As we traverse the plan tree from the leaf nodes to the root, we can calculate the cardinality distribution of an intermediate result, assuming that the cardinality distribution of the associated join operands is already available. Our algorithm is inspired by the approach of Chu et

al. [9]. More precisely, we assume that each distribution is approximated by a set of b alternative cardinality values with an associated probability (b is a tuning parameter, allowing to trade approximation precision for computational overheads). Each cardinality value represents a bucket, containing a range of cardinality values. Given cardinality values with associated probability values for each of the join inputs, we iterate over all combinations of input sizes. For each combination of input sizes, we approximate the associated Poisson distribution of the join output size (considering uncertainty in the selectivity of join predicates). Finally, we approximate the joint output distribution (integrating all possible input sizes) by a set of b representative cardinality values with associated probability values again.

5.2 Modeling Execution Costs for Plan Sets

ROME executes sets of complementary plans in parallel, terminating once the first such plan finishes. In Section 4, we introduce a simple proxy of plan diversity as a utility function for plan selections. However, the previously introduced model neglects several relevant aspects of the problem. First, it does not take into account the degree of uncertainty, associated with the sizes of specific intermediate results. As a tendency, joining many tables and using various join predicates creates more opportunities for estimation errors to propagate. Hence, selecting plans to safeguard against the most likely cardinality estimates is not possible with the prior model. Also, the prior model does not distinguish intermediate results that are likely to influence overall execution costs significantly from others. For instance, for small intermediate results, even a large relative estimation error may still have a limited impact on plan execution time as a whole (if other intermediate results in the same plan are significantly larger). We therefore propose a new model to guide plan selections. This model is based on the metric that we are ultimately interested in (rather than a proxy).

Definition 7 (Expected cost). *The expected (execution) cost of a plan set P is defined as the expected value of the minimal execution costs over all plans. More precisely, assume that the execution cost of plan $p \in P$ is modeled as random variable C_p, we judge plan sets by the quantity $\mathbb{E}(\min_{p \in P} C_p)$ (where $\mathbb{E}$ denotes expected value). Note that cost variables of different plans are not necessarily independent as plans may share the same intermediate results (meaning that an estimation error in one intermediate result affects multiple plans).*

For the purpose of plan selection, we model the cost of plans using the simple C_{out} cost model [10]. This metric simply sums up the sizes of intermediate results. While simplifying, this model has been shown to correlate well with cost functions of popular join operators [10], and experiments show that simple cost models are sufficient to identify efficient query plans [27]. Whereas different systems implement different operators, a cost model that is based on intermediate result sizes alone can be reused across a wide range of relational database management systems. Note that we use the aforementioned cost model only to select plans from plan candidates proposed by the query optimizer. On the other hand, the query optimizer uses a typically more precise cost model to select plan candidates, specialized to the characteristics of the associated execution engine.

We model the sizes of intermediate results as a discrete probability distribution. Hence, using the aforementioned cost metric, we can express the cost of plans as a discrete probability distribution as well. Our goal is to model dependencies between different plans that share intermediate results. Hence, rather than considering the cost of different plans independently, we must consider the executed combination of plans to calculate the expected execution costs.

Algorithm 2 shows the corresponding pseudo-code. As input, it obtains a set of plans to execute in parallel (P), as well as approximated size distributions for a *subset* of intermediate results. The output is the expected cost until the first plan in P finishes execution.

The second input requires further explanations. We approximate the size distribution of each intermediate result that appears in any plan using the method described in the previous subsection. However, when calculating the expectation of the minimum execution cost over all plans, we need to consider all possible combinations of intermediate result sizes. Even if the number of values per intermediate result is bounded by b (as explained in the previous section), the number of combinations still grows exponentially in the number of intermediate results we consider. To limit computational overheads, we introduce a second parameter that determines the precision of the approximation. This parameter, called k in the following sections, determines the number of intermediate results whose size we model as a distribution. For other intermediate results, we

Algorithm 2 Calculating expected value on the execution cost of the fastest plan in a plan set, executed in parallel.

```
1:  // Calculate the expected cost of the fastest plan in P, given size distributions D for a subset of intermediate
    results.
2:  function ExpectedCost(P, D = {⟨B_i, i⟩})
3:      // Collect all intermediate results in plans
4:      I ← ∪_{p∈P} I_p
5:      // Initialize size with point estimates
6:      S ← ∅
7:      for i ∈ I : ∄B_i : ⟨B_i, i⟩ ∈ D do
8:          S ← S ∪ {⟨i, M(i)⟩}
9:      end for
10:     // Create all combinations of sizes with associated probability
11:     C ← {⟨S, 1⟩}
12:     // Iterate over intermediate results with cardinality distribution
13:     for ⟨B_i, i⟩ ∈ D do
14:         C' ← ∅
15:         // Iterate over size combinations for prior results
16:         for ⟨S, p⟩ ∈ C do
17:             // Iterate over possible sizes of current result
18:             for ⟨s_i, p_i⟩ ∈ B_i do
19:                 // Expand sizes by adding size for current result
20:                 S' ← S ∪ {⟨i, s_i⟩}
21:                 C' ← C ∪ {⟨S', p · p_i⟩}
22:             end for
23:         end for
24:         C ← C'
25:     end for
26:     // Sum minimum cost for each combination, weighted by probability
27:     e ← 0
28:     for ⟨S, p⟩ ∈ C do
29:         e ← e + p · min_{p'∈P}(∑_{i∈I_{p'}} S[i])
30:     end for
31:     return e
32: end function
```

model them as point estimates when estimating execution costs. The input to Algorithm 2 therefore contains a set D of size distributions, associated with a subset of intermediate results, that should be considered in our approximation.

First, in Lines 6 to 9, Algorithm 2 iterates over all intermediate results for which no distribution is available. For those intermediate results, the algorithm models the size as the estimate by the query optimizer model (represented by $\mathcal{M}(i)$ in the pseudo-code). The corresponding estimates are stored in variable S, containing tuples that connect intermediate results to their size estimate. Next, the algorithm considers all possible combinations of intermediate result sizes. Those combinations will be stored in variable C. This variable contains tuples, pairing a size function (assigning intermediate results to size estimates) with a probability. The variable is initialized in Line 11, containing sizes for intermediate results without associated size distribution and a probability of one.

Next, Algorithm 2 considers intermediate results for which a size distribution is provided (as part of the function input). For each such intermediate result, the algorithm iterates over all size values with their associated probabilities. For each possible size value, the algorithm adds a new combination of size values by expanding any previously added combination (which does not yet consider the current intermediate

result). The probability for the expanded combination is calculated as the product between the probability of the original combination and the probability of the new size value. Again, this approach is a compromise between precision and computational overheads as it assumes that the sizes of different intermediate results are independent. After finishing iterations in Line 25, variable C contains a set of tuples, representing all possible combinations of intermediate result sizes, together with the associated probability.

Finally, Algorithm 2 calculates the expected value. To do so, it iterates over possible size combinations. The expression in Line 29 calculates the execution cost of each plan, given currently assumed sizes ($S[i]$ denotes the size assigned by S to intermediate result i). Then, it takes the minimal cost over all selected plans. The final result is the sum of those minima, weighted by the probability of the corresponding size combination. This value is returned as function result.

5.3 Greedy Plan Selection

We analyze the cost function introduced in the previous subsection. The purpose of adding more plans is to save execution overheads due to sub-optimal plan choices. We formalize cost savings next.

Definition 8 (Cost Savings). *Denote by $Cost(P)$ the expected minimal plan execution cost for plans P, calculated by Algorithm 2. We extend the definition of that function by assigning $Cost(\emptyset)$ to a high constant (higher than the execution cost of any plan). Then we can define cost savings $Save(P)$ of a plan set P as the improvement in execution cost, compared to the empty set: $Save(P) = Cost(\emptyset) - Cost(P)$.*

Due to the (expanded) definition of the cost function, it is clear that $Save(P)$ is non-negative. We prove monotonicity and submodularity next.

Theorem 5. *Cost savings are monotone.*

Proof. Since $Save(P) = Cost(\emptyset) - Cost(P)$, cost savings are monotone iff $-Cost(P)$ is monotone. Algorithm 2 calculates the cost as a weighted sum over minima, taking the minimum over different plans. All weights represent probability values and are therefore non-negative. Adding more plans can only decrease the minimum over different plans. As we sum over minima with non-negative weights, the cost function as a whole can only decrease when adding plans. This means that the negative of the cost function, $-Cost(P)$, can only increase when adding plans. That implies that cost savings are monotone. □

Theorem 6. *Cost savings are submodular.*

Proof. Cost savings are submodular iff $-Cost(P)$ is submodular. It is $-Cost(P) = \sum_s w_s(-\min_{p\in P}(C(p,s)))$ where w_s denotes the probability of size combination s and $C(p,s)$ denotes the cost of plan p, assuming sizes s. Assume $P \subseteq P'$ and p^* is a plan with $p^* \notin P \cup P'$. Consider the function $\mathcal{U}(P) = -\min_{p\in P} C(p,s)$ for an arbitrary but fixed s (assigning each intermediate result to a size value). It is $\min_{p\in P} C(p,s) \geq \min_{p\in P'} C(p,s)$ since $P \subseteq P'$. If $C(p^*,s) \geq \min_{p\in P} C(p,s)$ then adding p^* will not change the minimal value in any case, i.e., $\mathcal{U}(P \cup \{p^*\}) - \mathcal{U}(P) \geq \mathcal{U}(P' \cup \{p^*\}) - \mathcal{U}(P')$ holds and $\mathcal{U}$ is submodular. If $\min_{p\in P'} C(p,s) \leq C(p^*,s) \leq \min_{p\in P} C(p,s)$ then $\mathcal{U}(P' \cup \{p^*\}) - \mathcal{U}(P') = 0$ and $\mathcal{U}$ is submodular. If $C(p^*,c) \leq \min_{p\in P'} C(p,s)$ then $\mathcal{U}(P \cup \{p^*\}) - \mathcal{U}(P) > 0$ and $\mathcal{U}(P' \cup \{p^*\}) - \mathcal{U}(P') > 0$ but $\mathcal{U}$ is submodular again due to $\min_{p\in P} C(p,s) \geq \min_{p\in P'} C(p,s)$ and $\mathcal{U}(P \cup \{p^*\}) = \mathcal{U}(P' \cup \{p^*\})$. A weighted sum over submodular functions is submodular if the weights are positive. □

The properties of the cost savings function motivate a greedy algorithm that successively adds plans with optimal cost reduction (until the maximal number of plans is reached). This algorithm is similar to Algorithm 1 and therefore omitted due to space restrictions. We prove guarantees on the result quality, produced by the greedy algorithm.

Theorem 7. *Greedy optimization achieves cost savings within factor $(1 - 1/e)$ of the optimum.*

Proof. This follows immediately from the properties of the cost savings function (non-negative, monotone, and submodular) and the properties of the greedy algorithm [34]. □

Finally, we discuss how to trade computational overheads for approximation precision. Algorithm 2 depends on a set of intermediate results that are assumed to be variable (whereas the others are assumed constant). To

limit computational overheads, we select a limited number (k) of intermediate results to model via probability distributions. Modeling size as a distribution matters more for intermediate results that have significantly different sizes under different assumptions. Hence, after calculating an approximate probability distribution for all intermediate results, we select the k intermediate results with the largest gap between minimal and maximal size, represented in the associated distribution. For those intermediate results, we pass the size distribution as input to Algorithm 2.

5.4 Integer Linear Programming Approach

As an alternative to the greedy algorithm, we show how to transform plan selection into integer linear programming. The transformation differs from the one presented in Section 4.2 due to a different optimization objective. Now, our goal is to minimize expected costs. This influences the objective function as well as the required variables and constraints. We model intermediate result sizes using the same techniques as before. In particular, we assume that the size distribution of a bounded number of k intermediate results is approximated, using one discrete distribution per result with b values. This implies b^k possible combinations of intermediate result sizes. We denote by s one of those combinations in the following (we will also call them *scenarios*). By w_s, we denote the associated probability. Note that the set of possible combinations does not depend on the selected plans.

5.4.1 Variables. We introduce binary variables m_p for each plan $p \in P$. If $m_p = 1$ then the corresponding plan is selected for execution. Our goal is to minimize the expectation of the minimum cost over all selected plans (since query execution terminates once the first executed plan finishes). We introduce several auxiliary variables to model our cost objective. First, we introduce variables z_s for each size combination s, representing the minimal execution cost over all selected plans, assuming sizes s. Second, we introduce binary variables y_s^p (for each plan candidate p and size combination s), indicating whether plan p is the selected plan with minimal execution cost, given sizes s. More precisely, we set $y_s^p = 0$ iff plan p is the one with minimal execution cost in scenario s (and $y_s^p = 1$ iff the plan is not the one with minimal execution cost). Third, we introduce auxiliary variables v_s^p, representing the effective cost of plan p in scenario s, integrating the cost of the plan in that scenario as well as the decision whether the plan is executed in the first place. If the plan is not selected for execution, it is assigned to a high cost constant (ensuring that it will not determine the value of the minimum cost).

5.4.2 Constraints. At most n plans should be executed in parallel. We express this constraint using the following inequality: $\sum_{p \in P} m_p \leq n$. Next, we show how to constrain auxiliary variables. Variables v_s^p represent the effective cost of plan p in scenario s. We set $v_s^p = m_p \cdot C(p, s) + L \cdot (1 - m_p)$ where L is a large constant (e.g., a multiple of the cost of the most expensive candidate plan in the worst scenario). This equation ensures that the value of selected plans is set to their respective cost (in scenario s), whereas plans that are not selected for execution are assigned to a high value (guaranteed not to become the minimum). Variables y_s^p are set to zero if plan p finishes execution first in scenario s (thereby determining the overall cost of execution). We ensure that exactly one plan must be the first to finish, imposing $\sum_{p \in P} y_s^p = |P| - 1$ (we add one such constraint for each scenario s).

Next, we link variables y_s^p to variables z_s. We set

$$z_s \geq v_s^p - L' \cdot y_s^p \tag{1}$$

, introducing one such constraint for each plan p and scenario s. Here, L' is another large constant. The purpose of those constraints is to lower-bound the cost of the fastest plan in scenario s. If $y_s^p = 0$ then plan p is assumed to be the fastest plan in scenario s. In that case, execution costs in scenario s are indeed lower-bounded by the cost of that plan (expressed via variable v_s^p). We do not explicitly prevent y_s^p from taking value zero for plans not selected for execution. However, such value assignments do not optimize the objective discussed in the next subsection. If $y_s^p = 1$ then the corresponding plan is not the first to finish. In that case, the right side of Equation 1 takes on negative values (for an appropriate choice of constant L'). Because of that, the corresponding equation does not influence the lower bound on minimal plan cost.

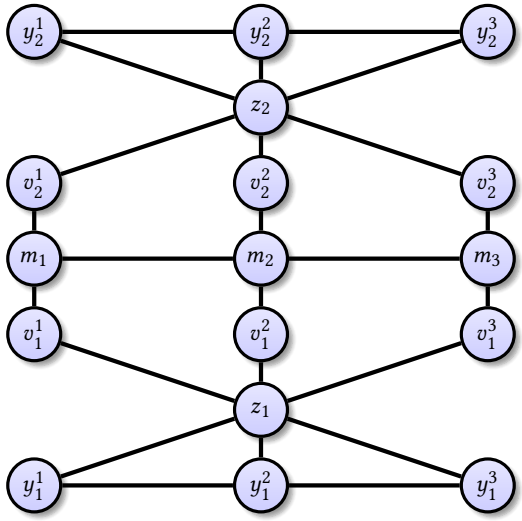

Fig. 4. Dependency graph of decision variables in ILP model via cardinality probabilistic model.

5.4.3 Objective Function. Our goal is to minimize the expected cost of parallel plan executions. Variables z_s represent parallel execution cost (i.e., cost until the first plan finishes) in scenario s. The expected value over all scenarios is obtained by summing up those variables, weighted by the probability of the corresponding scenario (which we denote by w_s). Hence, we minimize the linear function $\sum_s w_s \cdot z_s$.

Example 8. We re-use the situation depicted in Figure 2 to illustrate the ILP transformation. In this situation, we have three candidate plans. Hence, we introduce three binary variables m_1, m_2, and m_3, indicating whether the corresponding plan is selected. Assume that we associate one single intermediate result with a cardinality distribution (i.e., $k = 1$) and that we use two buckets for the approximation (i.e., $b = 2$). In that case, we have two possible scenarios in total. Hence, we introduce variables v_1^1 to v_2^3 to represent the effective cost values of the three plans in the two scenarios, as well as variables y_1^1 to y_2^3 to indicate which of the three plans finishes first in the corresponding scenario. We also introduce variables z_1 and z_2, representing parallel execution costs in each scenario. Figure 4 illustrates the variables, as well as the constraints connecting them. For instance, plan variables m_1 to m_3 are connected by constraints on the number of selected plans. Variables y_1^1 to y_1^3 and y_2^1 to y_2^3 are connected by constraints, making sure that only a single plan can finish first in each scenario. Variables z_1 and z_2 are connected to other variables, representing plan cost and the relative ranking of plans in the respective scenario.

6 COMPLEXITY ANALYSIS

We analyze the computational hardness of plan selection.

Theorem 9. *Selecting plans to maximize the number of intermediate results is NP-hard.*

Proof. We show NP-hardness by a polynomial-time reduction from maximum coverage. An instance of maximum coverage is defined by a collection of sets $S = \{S_1, S_2, \ldots\}$. The goal is to select at most n sets $S^* \subseteq S$ such that the union of their elements, $\cup_{s \in S^*} s$ has maximal cardinality. Given an instance of maximum coverage, we introduce one plan for each set $s \in S$. Also, the intermediate results associated with the plan represent the elements of the set s. In our scenario, we are interested in selecting no more plans than necessary (as doing so decreases the degree of parallelism per plan). This is expressed by a small penalty for each selected plan. For the transformation, we set the penalty to zero. Then, the optimal set of plans represents a solution to the original maximum coverage problem. □

Theorem 10. *Selecting plans to minimize expected parallel execution costs is NP-hard.*

Proof. We use a polynomial-time reduction from maximum coverage again. Denote by $S = \{S_1, S_2, \ldots\}$ the set of sets characterizing an instance of maximum coverage. Furthermore, denote by $U = \cup_{s \in S} s$ the universe of all elements contained in S. We associate each set $s \in S$ with a candidate query plan. Also, we associate each element in U with one scenario, i.e., with a combination of intermediate result sizes (using a suitable choice for the number of intermediate results and number of points for each cardinality distribution). For the scenario associated with a specific element $u \in U$, we ensure that all plans associated with sets containing u have a cost of zero in this scenario. All other plans have a cost of one in this scenario. Finally, we associate each scenario with equal probability. We select plans to minimize the weighted sum of minimal plan costs over all scenarios. The cost of each scenario is either zero (if at least one plan has cost zero in this scenario) or one (if no plan has cost zero in this scenario). The number of scenarios with a cost of zero corresponds to the number of elements covered. Selecting a set of n plans minimizing expected costs is therefore equivalent to selecting n sets to maximize coverage. □

Next, we analyze complexity of the proposed algorithms. We denote by x_p the number of candidate plans, by x_i the number of unique intermediate results, by x_l the maximal number of intermediate results per plan, and by n the number of plans selected for execution. For the probabilistic model, time complexity depends additionally on the number k of intermediate results associated with a cardinality distribution, and on the maximal number b of points considered per distribution. First, we analyze how the number of variables and constraints in the ILP formulation depends on the problem dimensions.

Theorem 11. *The ILP representing plan selection for maximizing diversity has a number of variables in $O(x_p + x_i)$ and constraints in $O(x_i \cdot x_p)$.*

Proof. We introduce $O(x_p)$ decision variables related to the selection of each plan. Also, we introduce one variable for each intermediate result (in $O(x_i)$). The number of constraints is dominated by constraints connecting plans to their intermediate results. Their number is in $O(x_p \cdot x_i)$. □

Theorem 12. *The ILP representing plan selection for minimal expected costs uses a number of variables and constraints in $O(b^k \cdot x_p)$.*

Proof. The number of scenarios (i.e., combinations of intermediate result sizes) is given by b^k. The number of variables is dominated by variables like v_s^p and y_s^p, introduced for each combination of plan and scenario. Hence, the number of variables is in $O(b^k \cdot x_p)$. The number of constraints is equally dominated by constraints on auxiliary variables that are introduced for each plan and scenario (e.g., calculating the values for auxiliary variables z_s^p). Hence, the number of constraints is also in $O(b^k \cdot x_p)$. □

Next, we analyze time complexity of the greedy algorithms.

Theorem 13. *Greedy plan selection via maximizing unique intermediate results has time complexity in $O(x_p \cdot n \cdot x_l)$.*

Proof. The greedy algorithm performs up to n iterations. In each iteration, it compares utility for up to x_p plans to select the optimal candidate. To evaluate a single plan, the greedy algorithm has to compare its intermediate results to the ones already covered by previously selected plans. We assume that covered intermediate results are stored in a data structure that allows containment checks in $O(1)$ (e.g., a hash set implementation). In that case, the overhead per iteration is in $O(x_p \cdot x_l)$ (where n_l denotes the maximal number of intermediate results per plan). Multiplying by the number of iterations (n) yields the postulated complexity. □

The greedy algorithm described in Section 5.3 requires a pre-processing step, associating all relevant intermediate results with a cardinality distribution (later, only a subset of those will be associated with a distribution for cost estimation). We approximate the cardinality distribution of intermediate results by discrete buckets and denote by b the number of buckets used.

Theorem 14. *Associating all intermediate results with cardinality distributions requires time in $O(b^3 \cdot x_i)$.*

Proof. We calculate an output size distribution, given input size distributions. The number of value buckets per distribution is limited to b. We iterate over all combinations of input buckets (b^2). For each combination, we consider b value buckets for the Poisson distribution associated with the combination of input sizes. Overall, calculating the output distribution for one intermediate results has time complexity in $O(b^3)$. This algorithm is invoked for each intermediate result. Hence, the total time complexity is $O(b^3 \cdot x_i)$. □

During greedy selection, only k intermediate results are associated with a distribution.

Theorem 15. *Greedily selecting plans to minimize expected cost has time complexity in* $O(n^2 \cdot x_p \cdot b^k \cdot x_l)$.

Proof. The greedy algorithm iterates up to n times. In each iteration, it compares up to n_p plan candidates. For each plan candidate, it calculates expected costs (Algorithm 2). Internally, the cost estimation algorithm iterates over all possible scenarios. Their number is in $O(b^k)$. For each scenario, the algorithm calculates the cost minimum of up to n plans. Calculating the cost of a plan requires summing over up to x_l intermediate results. Hence, the complexity of cost estimation is in $O(b^k \cdot n \cdot x_l)$. Multiplying by the number of iterations and cost comparisons per iteration yields complexity in $O(n^2 \cdot x_p \cdot b^k \cdot x_l)$. □

Note that caching cost values can reduce time complexity, in exchange for added space complexity.

7 EXPERIMENTAL EVALUATION

We describe the experimental setup in Section 7.1. In Section 7.2, we analyze the performance of different strategies for two benchmarks. In Section 7.3, we compare greedy algorithms against integer linear programming (ILP). Section 7.4 analyzes the impact of tuning parameters on performance. Section 7.5 analyzes other factors that contribute to ROME's performance. Section 7.6 is a case study, showing how our approach makes execution more robust for a specific query.

7.1 Experimental Setup

We use two benchmarks: join order benchmark (JOB) [27] and Stack. The database of the JOB benchmark consumes 6.7 GB, and the one associated with Stack consumes 61 GB (we did not create any indexes beyond the ones generated automatically by PostgreSQL). JOB is challenging to optimize due to skewed data. We evaluate all 113 queries in the benchmark. The Stack benchmark is a new real-world datasets created by Marcus et al. [30]. It contains over 18 million questions and answers from StackExchange websites (e.g., StackOverflow.com), collected over a period of ten years. In our experiments, we randomly select 10 queries over 11 SPJ templates (out of 16 templates in total), resulting in 110 queries in total.

We compare algorithms in terms of per-query and per-benchmark run time (we use the average of three runs), reporting 95% confidence bounds. We implement our approach on top of the PostgreSQL engine (PostgreSQL 12.11). The approach is implemented in Python 3.8.10. We use Gurobi 10 as ILP solver. Unless noted otherwise, we associate $k = 3$ intermediate results with a cardinality distribution, approximate using $b = 10$ buckets, and select $n = 3$ plans. All experiments use a 24-core server (two 2.30 GHz 12-core Intel(R) Xeon(R) Gold 5118 CPUs), running Ubuntu 20.04.4 LTS. The total DRAM size is 252 GB.

7.2 Comparison to Baselines

Figure 5 compares ROME to several baselines. All baselines use the same plan execution engine (PostgreSQL 12.11 [36]). For baselines executing a single plan, we increase the maximal degree of parallelism that can be exploited for execution to 24 (the number of cores on our target platform). We verified increasing the degree of parallelism for those baselines improves performance, compared to PostgreSQL's default settings, indeed. We use a timeout of 60 seconds per query and baseline. Baseline PG executes the plan proposed by the original PostgreSQL optimizer (i.e., this is the original PostgreSQL, exploiting full parallelism for single plans). AQO [21] is a PostgreSQL extension. For each input query, AQO tries out a sequence of query plans until convergence, using runtime statistics collected during the execution of prior plans to improve cardinality models used to select the next plan. We report only the time required for executing the last plan in the sequence in the following figures. This corresponds to an upper bound on the performance of similar baselines that perform trial runs before selecting a query plan. BAO [30] uses reinforcement learning to select between plans, suggested by the original query optimizer with different optimizer settings. To make the comparison fair, we

Table 1. Number of timeouts for different methods on the JOB and Stack benchmarks (averages over three runs).

	PG	PB/-3	BAO/-I/-R	AQO	LC-3	MPD	PM
JOB	1	1/1	0.67/1/0.33	1	0	0	0
Stack	5.67	9/9	8/9/9	7	6	0	0

consider the same optimizer parameters and parameter values as BAO for ROME (i.e., both approaches select from the same superset of plan candidates). BAO is trained at run time, based on the input queries (we do not assume that prior queries on the benchmark database are available for training). Hence, the order in which queries are observed may lead to different learned policies. Hence, we experiment with the original query order (BAO), the inverse order (BAO-I), as well as a random query order (BAO-R). PB is a re-implementation of the Plan Bouquets approach [13]. We use the pg_hint_plan tool[2] to change the selectivity estimates for up to three unary predicates, considering five uniformly spaced selectivity values per predicate and all associated combinations. This prompts the PostgreSQL planner to generate different plans for the same query. As in the original approach [13], we execute resulting plans, starting with the ones that assume low selectivity, using a series of exponentially increasing timeouts. As a variant, among the plans generated with different selectivity estimates, we execute the three plans with the lowest cost estimates in parallel (we verified that executing all plans in parallel performs worse). This baseline is reported as PB-3. Baseline LC-3 considers the same plan space as ROME, executing the three plans with the lowest cost estimates in parallel until the first one finishes. Oracle designates a baseline that executes the optimal plan for each query, considering candidate plans returned by the PostgreSQL optimizer for different configurations. We identify the optimal plan by executing all candidate plans. MPD (maximizing plan diversity) refers to the greedy algorithm and the model proposed in Section 4. PM (probabilistic model) refers to the greedy algorithm described in Section 5. We limit the number of selected plans to (at most) three. Note that the set of plans selected by both ROME versions does not necessarily include the default plan, selected by the original PostgreSQL optimizer.

Figure 5 compares the performance of all baselines for JOB and Stack. Compared to PostgreSQL (PG), PB and PB-3 (which performs better among the two) improve performance significantly for Stack, reducing total execution time for both benchmarks by approximately factor 2. BAO performs best in the original variant, improving performance compared to PG for both JOB and Stack. This results in a total time improvement of more than factor 2. AQO and LC-3 perform slightly better than BAO on Stack whereas improvements on JOB are not statistically significant. For both benchmarks, MPD and PM perform better than all other baselines except for the Oracle (which corresponds to the theoretical optimum). PM performs slightly better than MDP in both cases, reducing execution time by 10% for JOB. Table 1 reports on the number of timeouts (query execution time over 60 seconds), incurred by different approaches (we group different variants of the same approach and separate associated results using "/"). Again, MPD and PM achieve the best results among all approaches and are the only ones without timeouts for both benchmarks.

Figure 6 provides more details and reports speedups of PM, compared to BAO, for each query on the join order benchmark. Speedups of PM over BAO reach up to a factor of 19X (whereas BAO realized speedups of factor at most 2.4X over PM). Table 2 provides further details, reporting aggregate speedups for groups of queries, linked by the number of non-equality unary predicates (e.g., LIKE expressions) where selectivity estimation is particularly hard, especially considering correlations between predicates. Clearly, speedups increase if queries contain many such predicates. Finally, we examine the performance gap between LC-3 and ROME (i.e., MDP and PM). We hypothesize that using the same execution approach as ROME, while selecting plans with lowest cost, leads to less diverse plan sets (thereby increasing the risk of few cardinality estimation errors impacting all plans). We use the total number of distinct intermediate results over all queries as a measure of plan diversity. On JOB, we count 1804 for PM and 1625 for LC-3. On Stack, we count 1574 for

[2]https://github.com/ossc-db/pg_hint_plan/

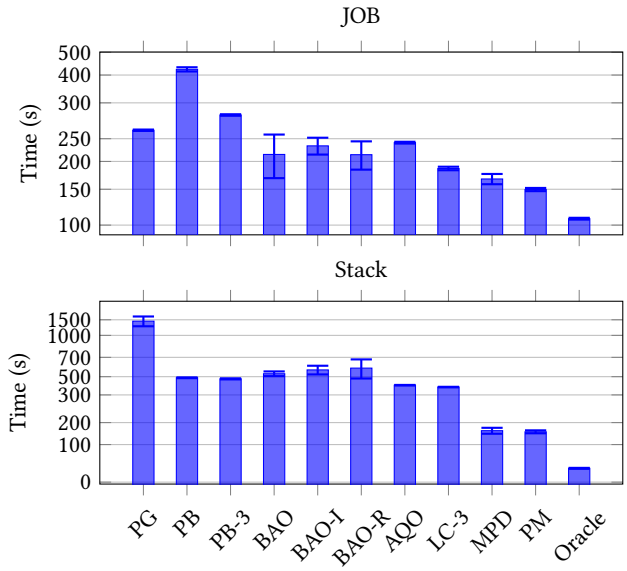

Fig. 5. Performance comparison (logarithmic y-axis).

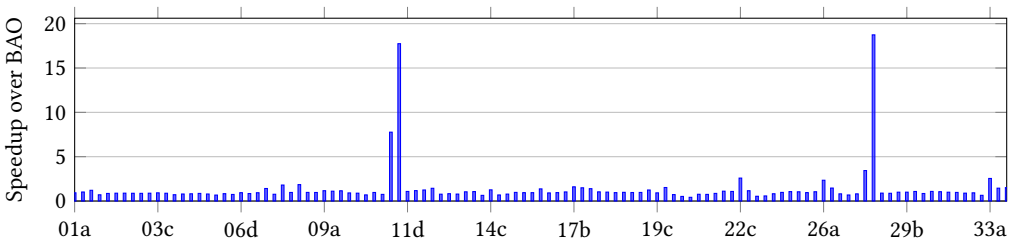

Fig. 6. Speedups of PM over BAO on queries of the join order benchmark.

Table 2. Relative performance of ROME compared to BAO, broken down by the number of error-prone predicates.

Nr. EP predicates	**0**	**1**	**2**	**3**
Avg. speedup	0.94	1.14	1.30	5.60
Nr. queries	17	23	68	5

PM and only 882 for LC-3, correlating with a larger performance gap on Stack. These results are consistent with our hypothesis.

7.3 Comparing ROME Variants

We introduced a greedy algorithm and an ILP transformation for both problem models. In this subsection, we compare both variants in terms of scalability and result quality. For that purpose, we consider additional optimizer flags (thereby increasing the search space to test scalability). Specifically, we consider all value combinations for six Boolean flags, influencing the behavior of the Postgres optimizer (specifically, we consider

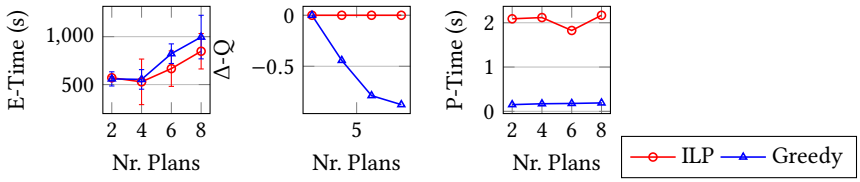

Fig. 7. Execution time, number of distinct results, and planning time when varying the number of selected plans for MPD model on JOB.

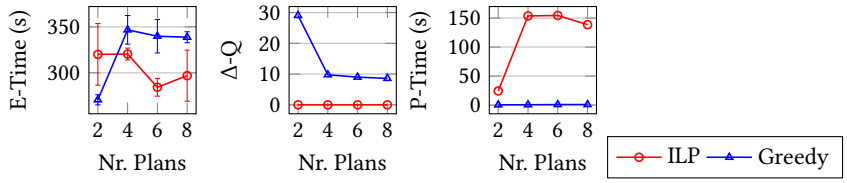

Fig. 8. Execution time, expected plan cost, and planning time when varying the number of selected plans for the probabilistic model on JOB.

the flags `enable_nestloop`, `enable_hashjoin`, `enable_mergejoin`, `enable_seqscan`, `enable_indexscan`, and `enable_indexonlyscan`). In total, we consider 64 value combinations. In the following experiments, we vary the number of plans selected for execution. As we will see, this parameter has significant impact in particular on optimization time.

Figure 7 compares greedy optimization to ILP for the utility model measuring the number of distinct intermediate results. While varying the number of executed plans, the greedy algorithm has slightly higher execution costs in average while the confidence bounds overlap (note that executing more than four plans in parallel does not in general improve execution costs). On the other hand, ILP achieves consistently a higher number of intermediate results. The gap grows in the number of selected plans. This means that ILP finds better solutions but, due to the simplicity of the first model, those improvements do not reliably translate into execution time gains. On the other hand, the greedy algorithm is significantly faster.

Figure 8 reports results for the probabilistic model (i.e., we minimize expected parallel execution costs). In this scenario, ILP achieves statistically significant improvements in run time. This shows that, using a more precise utility model, exhaustive optimization can yield advantages. Better execution times correlate with a consistent quality advantage of ILP over greedy, measuring solution quality by the probabilistic model. Planning time is significantly higher for ILP and increases as the number of selected plans increases.

In summary, ILP can be preferable over greedy when executing particularly expensive queries. However, ILP should be combined with the second (more precise) model to leverage its potential for finding optimal solutions.

7.4 Sensitivity Analysis

Our probabilistic model, described in Section 5, features two parameters, determining the precision at which cardinality distributions are approximated. In this subsection, we analyze how result quality as well as planning time depend on the settings for those parameters. More precisely, we consider the number of intermediate results that are associated with a cardinality distribution (k) and the number of buckets (b), used to approximate each distribution.

Figure 9 reports corresponding results. It reports planning time as "P-Time" and execution time as "E-time". Integrating more intermediate results into the approximation of the plan cost distribution (parameter k) has a moderate but consistently positive effect on run time. On the other hand, increasing the number of buckets above five has only a negligible impact on execution time. Planning time grows quickly in the number of

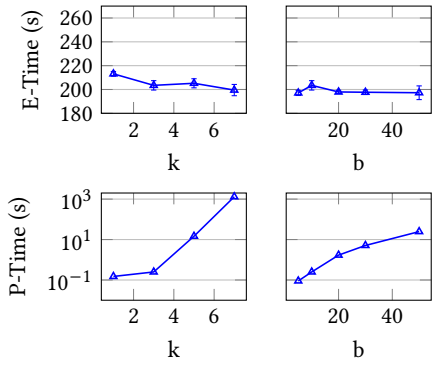

Fig. 9. Sensitivity analysis on JOB (logarithmic y-axis).

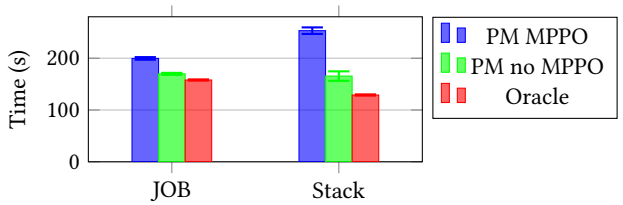

Fig. 10. Comparing performance of ROME with multiplan processing overheads (MPPO) to ROME without multiplan processing overheads (no MPPO) and performance of optimal plan (Oracle).

intermediate results whose cardinality distribution is considered. It also grows significantly when increasing the number of buckets.

In summary, the settings used in the previous experiments ($k = 3$ and $b = 10$) offer a good tradeoff between execution time and planning time.

7.5 Further Analysis

We analyze factors that contribute to ROME's performance.

Compared to an Oracle, selecting the optimal plan within our plan space for each query, ROME performs worse due to two factors. First, ROME may fail to select the optimal plan for parallel execution. Second, even if the optimal plan is selected, its performance may worsen due to executing multiple plans in parallel (as opposed to executing the optimal plan alone). Figure 10 quantifies both types of overheads. It compares the performance of ROME to the performance of ROME without multiplan processing overheads ("no MPPO" in the figure legend). We obtain results without multiplan processing overheads by executing each plan selected by ROME alone (without executing other plans concurrently), then using the minimum of all plan execution times for the same query. Finally, Figure 10 reports the time for executing the optimal plan (among all plans considered by ROME) as well. Averaging over both benchmarks, comparing execution times for "PPO" and "no PPO", executing multiple plans in parallel leads to relative overheads of 35%. On the other hand, failing to select optimal plans within ROME's plan space accounts for relative overheads of only 16%. Overheads due to parallel plan execution dominate.

Table 3 provides additional details on the impact of sub-optimal plan choices by ROME. It breaks down speedups of ROME, compared to using the original Postgres optimizer, by whether or not ROME selects the optimal plan within its plan space for execution (as one of up to three plans). For both benchmarks, ROME selects the optimal plan within its search space for about half of the queries (see columns JOB-P and Stack-P, reporting the percentage of queries for the corresponding benchmark). Average speedups (reported in columns JOB-S and Stack-S) reduce for queries where ROME fails to select the optimal plan. Still, ROME achieves

Table 3. Relative performance of ROME, broken down by whether or not the optimal plan is executed.

Optimal Plan	JOB-P	JOB-S	Stack-P	Stack-S
Included	44.2%	2.0	53.6%	8.3
Excluded	55.8%	1.3	46.4%	6.7

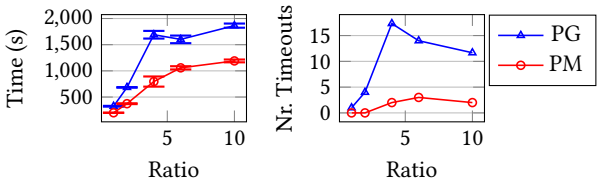

Fig. 11. Performance comparison of ROME and PostgreSQL when increasing the size of JOB.

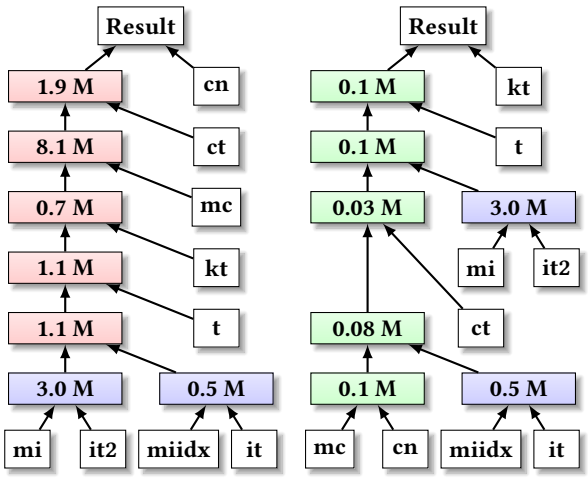

Fig. 12. Plans for JOB Query 13a with result sizes.

significant speedups in such cases as well, showing that even slightly sub-optimal plans can beat the optimizer defaults.

Finally, we study the impact of data size on the performance of ROME. Figure 11 compares the performance of ROME to the performance of PostgreSQL when increasing data size. More specifically, we duplicate rows in each table of the database associated with JOB (except for values in primary key columns for which we generate unique values when inserting rows). We increase the data size by up to one order of magnitude (values on the x-axis indicate how often rows are duplicated). Note that query plans are not fixed across scaling factors such that a larger amount of data does not necessarily result in higher processing overheads. Instead, having larger tables can in some cases change optimizer statistics and lead to better plan selections. For all data sizes, ROME achieves fairly stable speedups in the range between factors 1.5 and 2.1. This shows that ROME's speedups generalize to different data sizes.

7.6 Case Study

To illustrate the reason that our approach improves the selection of query plans, we consider Query 13a of JOB. In Figure 12, we present the query graph for two plans, chosen by our multiplan selection approach (the

left plan is, at the same time, the default plan selected by the PostgreSQL optimizer). In this visualization, white rectangles correspond to base tables, each labeled with a table alias. The rectangles with numerical labels represent join nodes. Blue rectangles indicate that the intermediate result is shared between plans. Red and green ones represent intermediate results that are unique to the plan in which they appear.

Clearly, the multiplan selection approach succeeds in selecting a set of plans with diverse intermediate results. Only a small number of intermediate results are shared across plans. Furthermore, the accumulated number of intermediate result tuples (a proxy for execution costs, which are more difficult to attribute to specific operators within the plan) differs significantly between the two plans. This happens despite the fact that both plans are considered optimal by the optimizer in certain parts of the plan search space. However, cardinality estimates for the plan on the left-hand side are overly optimistic, leading to the selection of a highly sub-optimal plan. For the plan on the right side, estimates are significantly closer to real values. This illustrates the benefit of diverse plan selections: using diverse intermediate results reduces the risk for systematic estimation errors that concern all executed plans at the same time.

8 RELATED WORK

ROME connects to a large body of prior work, aimed at making query optimization more robust. This includes prior research that focuses on improving the accuracy of size and cost estimation, e.g., by constructing histograms for selectivity estimation [7, 16, 23, 26], or learning size and cost models via machine learning [17, 24, 35, 38, 39, 43]. ROME does not aim at improving cost and size estimation. Instead, it relies on existing models and accounts for their inherent unreliability by selecting plans for parallel execution. ROME leverages plans proposed by the original optimizer, using different settings for optimizer configuration parameters. In that, it connects to recent work that leverages plans proposed by the original optimizer but learns optimizer hints via machine learning [11, 32], thereby biasing optimization towards subsets of the plan space. In contrast to that, ROME trades single-plan parallelism for executing multiple plans in parallel. In exchange, ROME does not require any prior training and no specialized hardware (e.g., prior work on learning-based optimizers often assumes that a GPU is available on the database server). Adaptive processing has been proposed as a means to make query execution more reliable. Corresponding approaches [1, 33, 41, 44] often use specialized execution engines or require an extension of existing engines. In contrast to that, ROME is completely non-intrusive and neither requires changes to the underlying engine nor access to its source code.

ROME selects complementary plans for parallel execution. In that, it relates to prior work on selecting plan sets [5, 12, 13, 19, 20], rather than single plans. However, prior work selects a single plan for execution or executes a sequence of plans for each query. ROME executes multiple plans in parallel and thereby exploits high degrees of parallelism, typical for current hardware platforms. ROME connects to prior work that focuses on robust query optimization [2, 3, 22]. The utility model presented in Section 5 relates to prior work associating intermediate results with probability distributions [2, 8, 9, 29], as opposed to point estimates. ROME differs by the way in which it uses those distributions, informing the selection of a plan set, rather than trying to select one single, robust plan. Broadly, ROME connects to prior work exploiting parallelism for query execution and optimization. This includes prior work on parallel query execution [4, 14, 18, 25] as well as parallel query optimization [15, 40, 42]. ROME exploits parallelism not for optimization but for execution. However, ROME does not parallelize the execution of a single plan. Instead, it executes alternative plans for the same query in parallel.

9 CONCLUSION

ROME (Robust Optimization by Multiplan Execution) exploits increasing degrees of parallelism on modern computing platforms to execute alternative plans in parallel. This approach may seem counter-intuitive since it guarantees redundant work for each query. However, the experiments show that investing parallelism to minimize the effects of costly optimizer mistakes pays off overall in scenarios where queries are not submitted concurrently. The proposed optimization framework is non-intrusive, does not require any extensions of the underlying database execution engine and avoids sequential evaluation of different plans (e.g., in the context of an adaptive strategy).

In future work, motivated by our experiments from Section 7.5, we plan to devise more efficient execution strategies for plan sets. This could lead to significant performance improvements, e.g., via work-sharing

between alternative plans for the same query. Also, we plan to consider mechanisms for early termination of plans that seem not promising, based on run-time statistics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Award No. 2239326.

REFERENCES

[1] Ron Avnur and Jm Hellerstein. 2000. Eddies: continuously adaptive query processing. In *SIGMOD*. 261–272. https://doi.org/10.1145/342009.335420

[2] Brian Babcock and S Chaudhuri. 2005. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*. 119–130. http://dl.acm.org/citation.cfm?id=1066172

[3] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *SIGMOD*. 107–118. https://doi.org/10.1145/1066157.1066171

[4] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *PODS*. 212–223. https://doi.org/10.1145/2594538.2594558 arXiv:arXiv:1401.1872v1

[5] P. Bizarro, N. Bruno, and D.J. DeWitt. 2009. Progressive parametric query optimization. *KDE* 21, 4 (2009), 582–594. https://doi.org/10.1109/TKDE.2008.160

[6] Peter Boncz, Angelos Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: Adding join crossing correlations with skew to TPC-H. *LNCS* 10661 (2018), 103–119. https://doi.org/10.1007/978-3-319-72401-0_8

[7] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1998. Random sampling for histogram construction. *SIGMOD Record* 27, 2 (1998), 436–447. https://doi.org/10.1145/276305.276343

[8] Francis Chu, Joseph Halpern, and Johannes Gehrke. 2002. Least expected cost query optimization: what can we expect?. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 293–302.

[9] Francis Chu, Joseph Y Halpern, and Praveen Seshadri. 1999. Least expected cost query optimization: An exercise in utility. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 138–147.

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

[11] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25. https://doi.org/10.1145/3588963

[12] Anshuman Dutt. 2014. QUEST : An exploratory approach to robust query processing. *PVLDB* 7, 13 (2014), 5–8.

[13] Anshuman Dutt and Jayant Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*. 1039–1050. https://doi.org/10.1145/2588555.2588566

[14] A Hameurlain and F Morvan. 2000. An overview of parallel query optimization in relational systems. In *Proc. DEXA*. 629–634.

[15] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. 2008. Parallelizing query optimization. In *VLDB*. 188–200. https://doi.org/10.14778/1453856.1453882

[16] Zhen He, Byung Suk Lee, and X. Sean Wang. 2008. Proactive and reactive multi-dimensional histogram maintenance for selectivity estimation. *Journal of Systems and Software* 81 (2008), 414–430. https://doi.org/10.1016/j.jss.2007.03.088

[17] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: Learn from Data, not from Queries! *PVLDB* 13, 7 (2019), 992–1005. https://doi.org/10.14778/3384345.3384349 arXiv:1909.00607

[18] Bhaskar Himatsingka, J Srivastava, and TM Niccum. 2007. *Tradeoffs in Parallel Query Processing and its Implications for Query Optimization.* Technical Report. 1–33 pages. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.2043&rep=rep1&type=pdf

[19] Arvind Hulgeri. 2004. *Parametric Query Optimization.* Ph. D. Dissertation. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.696&rep=rep1&type=pdf

[20] Arvind Hulgeri and S Sudarshan. 2003. AniPQO: almost non-intrusive parametric query optimization for nonlinear cost functions. In *VLDB*. 766–777. http://dl.acm.org/citation.cfm?id=1315517

[21] Oleg Ivanov and Sergey Bartunov. 2017. Adaptive cardinality estimation. *arXiv preprint arXiv:1711.08330* (2017).

[22] Harish D Pooja N Darera Jayant and R Haritsa. 2008. Identifying robust plans through plan diagram reduction. In *VLDB*, Vol. 24. Citeseer, 25.

[23] H. V. Kgadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Ken Sevcik, and Torsten Suel. 1998. Optimal histograms with quality guarantees. In *PVLDB*. 275 – 286.

[24] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: estimating correlated joins with deep learning. In *CIDR*. 1–8. arXiv:1809.00677 http://arxiv.org/abs/1809.00677
[25] M Kremer, J Gryz, and Others. 1999. *A survey of query optimization in parallel databases.* Technical Report. Citeseer.
[26] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung. 1999. Multi-dimensional selectivity estimation using compressed histogram information. In *SIGMOD*. 205–214. https://doi.org/10.1145/304181.304200
[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[28] Mike LePine. 2022. Poisson Distribution. In *College Statistics*.
[29] Yifan Li, Xiaohui Yu, Nick Koudas, Shu Lin, Calvin Sun, and Chong Chen. 2023. dbET: Execution Time Distribution-based Plan Selection. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.
[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2018. Neo: A Learned query optimizer. *PVLDB* 12, 11 (2018), 1705–1718. https://doi.org/10.14778/3342263.3342644 arXiv:1904.03711
[32] Tim Marcus, Ryan and Negi, Parimarjan and Mao, Hongzi and Tatbul, Nesime and Alizadeh, Mohammad and Kraska. 2022. Bao: Making Learned Query Optimization Practical. In *ACM SIGMOD Record*, Vol. 51. 6–13. https://doi.org/10.1145/3542700.3542702
[33] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proceedings of the VLDB Endowment* 14, 2 (2020), 101–113. https://doi.org/10.14778/3425879.3425882
[34] George L Nemhauser and Laurence A Wolsey. 1978. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of operations research* 3, 3 (1978), 177–188.
[35] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *SIGMOD*. 1017–1033. https://doi.org/10.1145/3318464.3389727 arXiv:1812.10568
[36] PostgreSQL. 2022. Group, The PostgreSQL Global Development. https://www.postgresql.org. Accessed: 2023-09-07.
[37] Li Quanzhong, Shao Minglong, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. 2007. Adaptively Reordering Joins During Query Execution. , 26–35 pages. https://doi.org/10.1109/ICDE.2007.367848
[38] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *PVLDB*. 19–28.
[39] Ji Sun and Guoliang Li. 2020. An end-to-end learning-based cost estimator. *PVLDB* 13, 3 (2020), 307–319. https://doi.org/10.14778/3368289.3368296
[40] Immanuel Trummer and Christoph Koch. 2016. Parallelizing Query Optimization on Shared-Nothing Architectures. *PVLDB* 9, 9 (2016), 660–671. https://dl.acm.org/doi/abs/10.14778/2947618.2947622
[41] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Transactions on Database Systems* 46, 3 (2021), 1–45. https://doi.org/10.1145/3464389
[42] Florian M. Waas and Joseph M. Hellerstein. 2009. Parallelizing extensible query optimizers. In *SIGMOD*. 871–882. https://doi.org/10.1145/1559845.1559938
[43] Lucas Woltmann, Claudio Hartmann, Maik Thiele, and Dirk Habich. 2019. Cardinality estimation with local deep learning models. In *aiDM*. 1–8.
[44] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*. 1721–1736.

Received October 2023; revised January 2024; accepted February 2024