Steering Query Optimizers: A Practical Take on Big Data Workloads

Parimarjan Negi[1], Matteo Interlandi[2], Ryan Marcus[1,3], Mohammad Alizadeh[1], Tim Kraska[1]
Marc Friedman[2], Alekh Jindal[2]
[1]MIT, [2]Microsoft, [3]Intel Labs
USA

ABSTRACT

In recent years, there has been tremendous interest in research that applies machine learning to database systems. Being one of the most complex components of a DBMS, query optimizers could benefit from adaptive policies that are learned systematically from the data and the query workload. Recent research has brought up novel ideas towards a learned query optimizer, however these ideas have not been evaluated on a commercial query processor or on large scale, real-world workloads. In this paper, we take the approach used by Marcus et al. in Bao and adapt it to SCOPE, a big data system used internally at Microsoft. Along the way, we solve multiple new challenges: we define how optimizer rules affect final query plans by introducing the concept of a rule signature, we devise a pipeline computing interesting rule configurations for recurring jobs, and we define a new learning problem allowing us to apply such interesting rule configurations to previously unseen jobs. We evaluate the efficacy of the approach on production workloads that include 150K daily jobs. Our results show that alternative rule configurations can generate plans with lower costs, and this can translate to runtime latency savings of $7-30\%$ on average and up to 90% for a non trivial subset of the workload.

ACM Reference Format:
Parimarjan Negi[1], Matteo Interlandi[2], Ryan Marcus[1,3], Mohammad Alizadeh[1], Tim Kraska[1] and Marc Friedman[2], Alekh Jindal[2]. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457568

1 INTRODUCTION

Cascades-style query optimizers [7] are popular in both commercial and open source database systems, e.g., Spark [31], Calcite [2], Greenplum [25], Snowflake [4], F1 [23], SQL Server [17], and SCOPE [3]. At its core, they have a set of rules that are used to enumerate all valid query plans. Each plan is assigned a cost, using a cost model and estimated cardinalities for the intermediate results in the plan. The lowest cost plan is chosen by the optimizer for execution. There are multiple classes of errors an optimizer can make, such as cardinality mis-estimates, inaccuracies in the cost model [13], and other poor heuristics. These mistakes can have a large adversarial influence on the low level decisions made by the optimizer, such as the choice of operator implementation and the join orders. For instance, severe cardinality underestimates can lead an optimizer to pick a disastrous plan involving nested loop joins [14].

	Workloads			
	A	**B**	**C**	**Total**
# Jobs	95K	15K	40K	150K
# Unique Templates	48K	10.5K	22K	80.5K
# Unique Inputs	29K	9K	18.5K	56.5K
# Unique rule signature	13K	837	2.5K	16.337K

Table 1: Production workloads used through the paper.

In recent years, several works have tried to solve such problems using machine learning: e.g., learned cardinalities [11], learned cost models [24], and even learned query planners [12, 14, 15]. The Bao [14] approach, in particular, handles the numerous optimizer pitfalls by limiting the search space of the query optimizer based on a given query. This is achieved by disabling a subset of the optimizer rules[1] that control how query plans are transformed, e.g., which operators, or algorithms, are used by the optimizer. Bao can learn, for example, that certain kinds of queries systematically have large underestimates, and decide to turn off nested loop joins for other similar queries. As a result, we can build sophisticated machine learning models to deeply influence the query plans *without* making deep changes in the query optimizer.

This paper fills the gap between recent research advances in steering a query optimizer, and the practical realities of industry strength query optimizers. Specifically, we consider the SCOPE [3] query engine used for petabyte-scale big data processing at Microsoft, analyze the current state of the SCOPE query optimizer, and apply the ideas in Bao to production workloads in SCOPE. While doing this, we solve multiple new challenges. We introduce the notion of a *rule signature* for a query, i.e., the set of rules that impact the final query plan output by the optimizer. The rule signature has been central to all of our heuristics for discovering interesting rule configurations. Even more importantly, it proves to be an excellent signal to cluster very different queries into groups where similar rule configurations lead to improvements. Intuitively, the rule signature compactly captures the code path that the query takes inside an optimizer. Furthermore, we devise a pipeline that can automatically extract interesting configurations from historical jobs, which can then be used online to improve future recurring jobs. Finally, we establish a learning problem, and provide some initial results on how machine learning can be used to improve previously unseen jobs by utilizing the results of our offline pipeline.

[1]Several query optimizers have their rules exposed as knobs for expert users.

Work done while Parimarjan Negi was at Microsoft.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457568

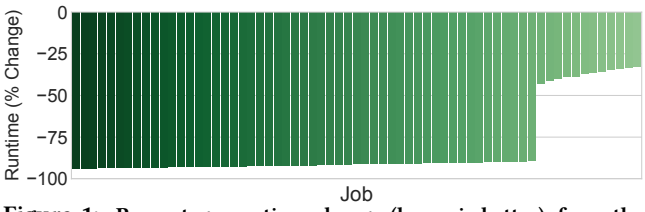


Figure 1: Percentage runtime change (lower is better), from the default rule configuration, for 65 productions jobs in Workload A.

We evaluate the efficacy of our approach on three workloads sampled from production clusters over a window of few weeks. For any single day, these include 150K SCOPE jobs (Table 1), comprising of 20K hours of total processing time, and spanning over 500 million containers. These jobs are also far more complex than the workloads used when evaluating Bao or several other learned query optimization approaches [11, 14, 15, 18], with up to hundreds of operators per SCOPE job. Furthermore, there is a massive search space of billions of valid rule configurations in contrast to 48 PostgreSQL configurations considered in Bao.

We find robust evidence, across the different workloads, that steering the search space of the SCOPE optimizer by choosing different rule configurations can lead to significant improvements in terms of execution times. Moreover, we can reliably identify templates for recurrent jobs running across many days, where certain rule configurations can lead to consistent improvements. To illustrate this, Figure 1 shows 65 jobs over a span of one week in Workload A, where the same rule configuration improves each query's execution time by hours, ranging from 50% to 90% faster runtimes. Overall, our analysis on the three production workloads suggests that there is a potential for 10−30% runtime improvements, on average, in almost 10 − 20% of the jobs that run for more than 5 minutes, which is the critical, resource intensive component of the workload.

Contributions. Summarizing, our key contributions are:

(1) **rule signature.** We analyze the usage of rules in SCOPE query optimizer, and introduce the idea of a rule signature for SCOPE jobs. rule signature is promising both to featurize and to cluster jobs, as it compactly captures useful information about the code path inside the optimizer.
(2) **Discovering interesting rule configurations.** We create a heuristics driven pipeline that uses recompilation and selective re-execution of query plans to discover interesting rule configurations. We provide robust evidence, and intuition, for why different rule configurations can improve performance in SCOPE. These techniques do not rely on our existing knowledge about these rules, and could potentially be scaled up to hundreds of more rules, or other configuration flags and settings. Moreover, we identify recurring queries across multiple weeks in which these new rule configurations can provide runtime improvements.
(3) **Learning.** In the more challenging cases, the same rule configurations can also cause regressions for unseen queries. We therefore formulate selecting rule configurations at compile time as a learning problem, and present results in which we use a learned model to choose rule configurations for new queries. These results use only a small subset of the total workload, but our results shows that it can scale over full production workloads.

Organization. The remainder of the paper is organized as follows: In Section 2, we describe related work, in particular the relevant features of the Bao system our approach is based on. In Section 3 we give background on SCOPE, in particular we describe its rule-based optimization process. In Section 4, we describe the challenges in adapting a Bao like system for SCOPE, and in Sections 5, 6, and 7 we describe our approach for solving these challenges. The papers ends with conclusions and future work in Section 8.

2 RELATED WORK

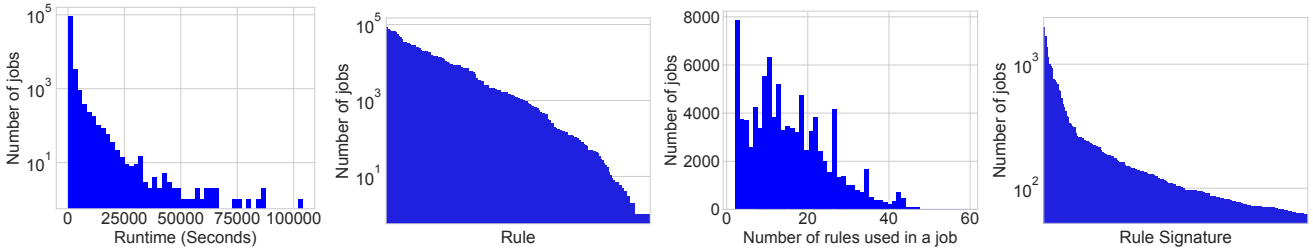
Learned query optimization. Many recent works use learned models to improve cardinality estimation [5, 6, 11, 18, 19, 27, 29, 30], which is a core component of query optimization. Other approaches [12, 15, 16] focus on directly generating query plans by learning from the output of the cost model or the runtimes. Our approach is based on Bao [14], and uses existing optimizer infrastructure to better explore, and choose from the query plans already being considered by the optimizer. Conceptually, this is similar to past work exploring the space of query plans, such as Picasso [8], or designing robust cost estimates for choosing query plans [26].

Bao. Marcus et al. [14] devised Bao as a system that leverages PostgreSQL query hints to generate 48 hint sets (or rule configurations). Each hint set essentially is like a simpler versions of the PostgreSQL query optimizer. Each simple optimizer disables a subset of the PostgreSQL flags. The hints affect the optimizer behaviour for choosing scan operators, join operators, and join orders. They treat each simple optimizer as an arm in a multi armed bandit problem. Given a new query, the model learns to choose one of the 48 arms. It is modeled as a reinforcement learning problem in which Bao sees a sequence of queries, and over time learns to make the correct decisions. Bao was evaluated on a custom dataset with queries ranging from a few to several hundred seconds. Bao did not rely on the PostgreSQL cost model. Instead, a core component of their system is a tree convolutional neural network which learns a cost model for tree structured PostgreSQL query plans by executing the plans.

SCOPE. The overall SCOPE design is described in Chaiken et al. [3]. Jindal et al. [9] give an overview of the Peregrine infrastructure used to introduce workload optimizations in SCOPE. There have been several efforts trying to introduce learned components in SCOPE. Sen et al. [22] uses learned models to automatically choose the number of concurrent containers a job should use. Wu et al. [28] developed a novel way to do cardinality estimation tailored to the SCOPE workloads. Siddiqui et al. [24] analyzed the new challenges for cost models in a cloud-based execution framework, and proposed a learning approach for SCOPE.

3 PRODUCTION QUERY PROCESSOR

In this section, we first describe the SCOPE query processor, and the workloads seen by it in production. Then, we analyze the optimizer rules in SCOPE, and finally we discuss our key requirements when applying ML to navigate the space of SCOPE optimizer rules.



(a) Histogram of SCOPE job runtimes.

(b) Number of jobs using each of the rules.

(c) Histogram of number of different rules used in a job.

(d) Number of jobs for 200 most common rule signatures.

Figure 2: Distributions of runtimes and queries during one day in Workload A.

Category	#Rules	#Unused Rules	Rule Examples
Required	37	9	EnforceExchange, BuildOutput GetToRange, SelectToFilter
Off-by-default	46	36	CorrelatedJoinOnUnion1, GroupbyOnJoin
On-by-default	141	37	NormalizeReduce, CollapseSelect, SelectPartitions, SequenceProjectOnUnion
Implementation	32	4	HashJoinImpl1, JoinToApplyIndex1, UnionToVirtualDataset

Table 2: Rule categories with some statistics and examples.

3.1 SCOPE Overview and Workloads

SCOPE is a large scale distributed data processing system. It powers production workloads from a range of Microsoft products processing petabytes of data every day [3]. SCOPE uses a SQL-like scripting language that is compiled into Direct Acyclic Graphs (DAGs) of operators. SCOPE scripts contain a mix of relational and user defined operators (in C# and Python). Since SCOPE scripts contain a data flow of multiple SQL statements, they are also referred to as *jobs*.

The SCOPE optimizer is structured very similarly to traditional cascades-style query optimizers: it transforms a logical query plan using multiple tasks in a top-down fashion. However, the SCOPE optimizer also makes all the decisions about how to partition the inputs, and selects the optimal amount of parallelism given the number of containers available for the job. The number of concurrent containers used by each job is referred to as *number of tokens* in SCOPE. The SCOPE optimizer estimates the cost of an operator to capture its runtime latency using a combination of data characteristics and other heuristics tuned over the years. For any query plan (or subplan), the cost of an operator is recursively combined with all of its children's cost, and ultimately the optimizer picks the plan with the cheapest total cost for execution.

3.1.1 SCOPE Workload Characterization. A large portion of SCOPE workloads consist of *recurring jobs*, i.e., periodically arriving templates with different inputs and predicates, that are part of the workflows of internal customers. These are often used to cook large volumes of raw data, run data mining or other analytical tasks, and populate dashboards for interactive analysis. The input *data streams* for these jobs can change daily. Recurring jobs belonging to the same template can be identified by discarding all variable values (e.g., predicate filters) and by computing the hash of the remaining information in the query graph.

Short running jobs v/s long running jobs. In Figure 2a we show the distribution of runtimes in Workload A for a single day. The shortest jobs take just a few seconds, but there are several that run for hours. Similar results also hold for the other workloads. Unsurprisingly, the shorter running jobs also consume fewer resources. We find that only about 10% of the jobs last over five minutes, but these consume 90% of the total containers used in the workload. Furthermore, distributed processing often leads to variance in the runtimes for the same query plans (unpredictability in resource allocation, hot spots due to cluster conditions, or large chain of data dependencies). Incidentally, short running jobs also have larger variance: for instance on Workload B we observed a variance of around 10%. Therefore, we focus on long running jobs in this work.

3.1.2 Metrics. Job runtime is the typical metric used to evaluate performance. However, in SCOPE, the following other metrics are equally important, particularly for monitoring the cost in terms of resource usage. Typically, many parallel jobs are being executed on the same set of resources, thus improving the utilization for any resource can help improve performance of other jobs, and reduce the load on the servers.

(1) **Runtime.** The total wall clock latency of executing the job from start to finish. This does not include the time a job may spend in a queue being scheduled.
(2) **CPU time.** This is the total CPU time across all vertices in a SCOPE job and indicates the computation cost of the job, and is useful to measure the CPU utilization in our clusters.
(3) **Total I/O time.** This includes data read, written, or copied to different containers and indicates the time spent on I/O.

3.1.3 A/B testing Infrastructures. The SCOPE infrastructure also provides A/B testing capabilities to evaluate the performance impact of new features in the SCOPE engine, e.g., during a new SCOPE release. This feature can also be used to execute jobs with different configurations, and compare performance. The A/B testing infrastructure can re-execute recent production jobs using production datasets but with outputs redirected to a dummy location [1, 10, 22, 28]. We use this A/B testing infrastructure for all the experiments in this paper. For all the reported execution times (or

other metrics when applicable) we re-executed the original production query plans and the alternate query plans (with different rule configurations) on the production clusters, and with the same set of available resources (50 tokens each for every job). In production, each job executes with different resources, as specified by the customer, but we typically find that improvements with a fixed set of resources also translate to improvements when more resources are used.

3.2 Analyzing SCOPE Optimizer Rules

We now analyze the query optimizer rules in SCOPE. There are 256 rules in the SCOPE optimizer that govern the algorithms and operators used to optimize a SCOPE job. These cover a wide range of standard optimizer rules: rewrite rules, rules for particular operators, join order and implementation rules and so on. It also involves operators and algorithms for SCOPE specific operators, like UNION ALL [3]. Figure 2b shows the distribution of how frequently each rule is used in Workload A jobs, and Figure 2c shows the distribution of how many different rules are used in a single job. Interestingly, even though $100 - 150$ rules are used frequently in the workload, typically only $10 - 20$ different rules are used in a single job. We divide the kind of rules used in SCOPE into four informal categories.

(1) **Required rules.** These rules are necessary for correctness in query processing, therefore we do not treat these as part of the learnable rule configurations. Examples include EnforceExchange or BuildOutput, which have no alternative rules.
(2) **Off-by-default rules.** These are rules that are either experimental, or unsafe due to extreme sensitivity to mis-estimates in the cardinalities. An example would be a collection of slightly different CorrelatedJoinOnUnion rules. These rules chooses to push UNION ALL operators under the JOIN operator, i.e., choose to do unions on each distributed node, and then combine them on a single node for the join, versus joining on each distributed node, and applying the union operator on the combined results. The performance of this rule can be extremely sensitive to the sizes of the intermediate results.
(3) **On-by-default rules.** These include most optimization rules and algorithms. Examples include various rewrite rules, join order rules, aggregation and sorting rules.
(4) **Implementation rules.** These rules are about the physical implementation of logical operators, such as JOIN or UNION ALL. For each operator type, one of the implementation rules must be enabled. For simplicity, we treat these as a single class of rules in this paper.

Table 2 summarizes some of the statistics about each category in the $95K$ jobs from one day in Workload A. Interestingly dozens of on-by-default rules are never used in this workload.

Definition 3.1. **Rule configuration.** We define a bit vector specifying whether each rule is enabled or disabled when optimizing a given job as the *rule configuration*. The default rule configuration in SCOPE has 46 rules which are disabled (off-by-default rules), and the rest are enabled. Only the enabled rules can be used by the optimizer. SCOPE exposes flags, or "hints" that allow end users to specify which rules should be enabled or disabled when optimizing a job, thus modifying the rule configurations can be easily done in SCOPE.

There are rules, such as some rewrite rules, which are just not applicable to a job (e.g., because the target operator is not part of the query). In other cases, there are rules that were not used because an alternative rule was used (e.g., JOIN operator implementation rules). To track this, we modified the SCOPE optimizer to log which rule contributes to any component of the final query plan.

Definition 3.2. **Rule signature.** We define a bit vector specifying which rules directly contribute to the final query plan produced by the optimizer as the *rule signature*. We refer to them as on rules if it is 1 in the bit vector, and off otherwise. For instance, consider a scenario with 10 total rules. A given query will be optimized using a particular rule configuration of the 10 rules, for instance, 1111111110 — which implies that the last rule is disabled, and the rest are enabled. Not all the enabled rules will impact the optimization process — for instance, some rules may just not apply for this particular query. Suppose only the first and the second rule was used during the optimization. Then, the rule signature of this query, with the given rule configuration, will be 1100000000. We refer to the rule signature of a query optimized using the default rule configuration as the *default rule signature*.

Figure 2d shows the distribution of the default rule signature of the jobs on one day in Workload A. Even though there are exponentially many possible rule signatures, in practice we observe that there is a lot of structure in the distribution of rule signatures. For instance, there are several rule signatures with almost 1000 jobs mapping to them every day.

3.3 Learning Requirements

Introducing any learning component in a complex and widely used system such as SCOPE is challenging. Below we discuss some of our requirements.

- **Domain knowledge vs full control.** An alternative approach to Bao, Neo [15] seeks to learn the complete optimizer, or a full component of it, from scratch. It benefits from having full control over the optimization decisions, but it can't use decades of carefully crafted and domain specific optimizations used in commercial optimizers such as SCOPE. Bao, and our adaptation in this paper, utilize all the existing knowledge and implementation in the optimizers.
- **Non-invasive.** The SCOPE optimizer has an extremely mature and large code base. It is very challenging to make big changes to the internals of the optimizer and include learning in specific subsystems while avoiding unwanted side effects. Our approach learns to effectively use knobs that are already exposed by the optimizer.
- **Ease of deployment as "plan hint".** It is always hard to deploy learning based approaches that may cause surprising regressions. One way to deploy such a system is to suggest the new rule configurations to the customers responsible for particular workloads. In fact, while rule flags are already

available and often used by customers, new rule configurations can be simply surfaced as an extension of this capability.

- **Scalability.** New code paths are constantly added to the optimizer, leading to ever increasing complexity. Besides rules, the optimizer has hundreds of other configurable flags and code paths. Our approach is designed such that it can potentially scale up to all available flags.
- **Focus on relevant metrics.** Several recent works focus on improving the cost model and cardinality estimation components of a query optimizer [11, 18, 19, 29, 30], including work on SCOPE [24, 28]. These remain challenging problems, and it can be hard to fix these errors in a way that improves the overall query optimization process. More importantly, improving an individual component, such as cardinality estimation, does not guarantee that the generated plans get better. In contrast, our learned approach attempts to directly optimize the metric we care about (e.g., execution time, I/O).

4 STEERING THE QUERY OPTIMIZER

In this section, we describe the problem of steering the query optimizer using rule hints, discuss the challenges we see in SCOPE, and give an overview of our approach to address those challenges.

Problem Statement. Given a new SCOPE job, the SCOPE optimizer always uses the default rule configuration to generate the query plan. Our goal is to output an alternative rule configuration which is better for optimizing this particular job, and for a given metric, such as runtime latency.

While the above problem statement is the same as in Bao [14], we discovered several novel challenges when tackling it for SCOPE. We discuss them below.

(1) **Large space of rule configurations.** We consider rules in SCOPE that affect a much wider range of optimizer behaviour than Bao. SCOPE has 219 non-required rules, thus there is a 2^{219} theoretical limit on the number of configurations. Many of these may not compile successfully due to implicit dependencies, but there are still billions of valid rule configurations (in contrast to 48 rule configurations considered in Bao). Therefore, we need a way to come up with a candidate set of rule configurations which may be interesting to explore further. At the same time, the large space of possibilities also make it more likely to discover interesting behaviors.
(2) **Expensive Executions.** It is prohibitively expensive to collect data about execution times for too many rule configurations. Therefore, we will not have enough data to learn a cost model, as in Bao. Instead, we need heuristics to choose which rule configurations should be executed.
(3) **Formulating the learning problem.** Due to the number of configurations, a multi-armed bandit problem does not scale up to a system like SCOPE. SCOPE query graphs are DAGs with up to hundreds of operator nodes, thus a graph based featurization scheme, as in Bao, is not directly applicable.

Overview of our approach. To overcome the above challenges, we follow four main steps:

(1) **Which jobs and rule configurations should we look at?** We select a subset of representative jobs to analyze. For each of them, we use heuristics to generate up to 1000 candidate rule configurations. (Section 5).
(2) **Can rule configurations improve runtimes?** We use heuristics to select 10 candidate rule configurations to execute and find configurations that lead to improved job runtimes. (Section 6.1, 6.2).
(3) **Extrapolate to other jobs.** We take the configurations that improved runtimes and extrapolate to other jobs across multiple days and templates which may benefit from these configurations. (Section 6.4).
(4) **Learn.** We collect run time data for jobs across multiple days, and train a supervised learning model to choose rule configurations. (Section 7).

The pipeline described above runs offline by utilizing SCOPE's compiler, flags, and A/B testing infrastructure to analyze past workloads. We can then use the learned models in an online scenario to either automatically use different rule configurations for new jobs, or to recommend new configurations to the customer.

5 DISCOVERING RULE CONFIGURATIONS

The naive approach would be to consider the exponentially many valid rule configurations for every job. Clearly, this is infeasible. Therefore, our goal is to adaptively discover M interesting rule configurations for a given job. Then, we recompile the job with each of the M rule configurations and analyze the generated query plans to find plans that are worth executing.

5.1 Job Span

Intuitively, we want to find rule configurations that can lead to interesting changes in the optimized query plan, while not exploring too many unworthy configurations. That is, we only want to explore configurations enabling/disabling rules that have an impact on the final query plan. Disabling a rule that can not impact the query plan will not make a difference (e.g., a rule that optimizes the GROUP BY operator for a job that does not have a GROUP BY clause). We describe how we prune the search space over the rules by means of a simple heuristic.

Definition 5.1. **Job span.** Given a job, its *span* contains all non-required rules which, if enabled or disabled, can affect the final query plan.

Generating a job span for optimizers, such as the SCOPE optimizer, where rules can have complex data-driven dependencies is challenging. Algorithm 1 shows the heuristics we use to approximate the job span. We already know that all the on rules in the default rule signature can impact the final query plan. Therefore, if we disable some of these rules, the optimizer may use some other rules instead. This algorithm seeks to find such alternative rules by iteratively disabling all rules that were used when optimizing a job and recompiling the job each time to see which new rules start getting used instead.

Limitation. The above algorithm does not capture all the possible rules that could impact the final query plan because complex dependency structures may be present in the rules, but are not indirectly observable using our heuristic. For instance, consider

Algorithm 1 Approximating query span

Input: Q {Query}
Output: *span*

```
span ← {}
config ← {1, 2, 3..., 220}        {all rule ids w/o required rules}
new_rules ← true
while new_rules do
    signature ← COMPILE (Q, config) {gets the rule signature for a query compiled with a given config.}
    on_rules ← GET_ON_RULES (signature) {finds all the rules in the signature bit vector that are 1}
    if on_rules == ∅ then
        new_rules ← false
    else
        config ← config \ on_rules
    end if
end while
```

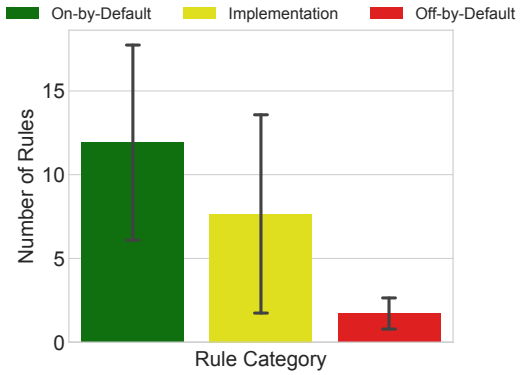


Figure 3: Average number of rules (±1 std.) in the span for each job in a day of Workload A, grouped by each category.

three rules A, B, C in which B and C are alternative rules, but both have a dependency on A, such that they can only be used if rule A is used. Let's say that with all rules enabled, A and B are in the rule signature. In its first iteration, the heuristic algorithm computes the job spans by disabling both rules A and B. Thus, it will fail to discover that rule C can impact the final plan as well. While to address this shortcoming, we need more detailed knowledge about the rules, and their dependencies, we can already find many interesting rule configurations using our heuristic despite missing these complex dependencies, as we show in our analysis in Section 5.3.

5.2 Configuration Search

Relative to the total number of rules, most queries have much fewer rules in their span. For instance, Figure 3 shows the average, and standard deviation of the number of rules in the span of each job on a single day on Workload A. We group these rules by the rule categories described earlier. On average we see that only up to 20 rules among the 219 non-required rules are used by each job. This reduces the search space for rule configurations considerably.

Assuming independence of rule categories. Informally, two rules can be considered independent if enabling or disabling one of them does not impact the behaviour of the other rule in a query optimizer. This may be because they apply to different parts of a job. Intuitively, there must be a lot of rules in a query optimizer which are usually independent: for instance, particular join implementation rules are probably independent of certain rewrite rules. Knowing subsets of rules are independent can dramatically reduce the search space of rule configurations. For instance, consider 5 rules. There are $2^5 = 32$ rule configurations. If we can establish that there are two groups of rules with two and three rules such that the groups are independent of each other. Then, we will need to only explore $2^2 + 2^3 = 12$ rule configurations. While it is hard to formally establish, and discover such independent subset of rules, in this work we make the assumption that each category of rules are independent of the other categories. In practice, this makes exploring the space of rules much easier.

Randomized Configuration Search. We use randomized search to enumerate M candidate configurations. For a given job, the list of rule configurations are generated by:

(1) Enable all the rules that are not in the span of the given job. [2]
(2) For each rule category, independently sample a subset of rules from the job span. Disable these rules, and enable all others. This gives us a new rule configuration.
(3) If the rule configuration has not been seen before, add it to the candidate list. Repeat until M configurations are generated.

5.3 Recompilation Results

Selecting jobs to analyze. We run our initial analysis on jobs on one day from Workload A, Workload B, and Workload C. We filter out jobs that are faster than five minutes and longer than one hour. We avoid the short running jobs because the runtime variance makes it extremely hard to discern improvements between alternative query plans. We avoid the long running jobs because re-executing many alternative plans can take really long. From the remaining jobs, we select a random sample of $10-20\%$ queries from each workload. We generate up to 1000 unique rule configurations for each job to recompile. Surprisingly, for most queries, some of the recompiled plans with the new rule configurations have lower estimated costs. Figure 4 shows examples of queries from Workload A with the default cost, and the recompiled costs for each rule configuration. *This appears paradoxical because a cascades style optimizer like SCOPE guarantees that it finds the lowest estimated cost plan in the search space.*

Why does the optimizer find lower cost plans with different configurations? There are a few subtle ways by which changing rule configurations can impact the way estimated costs are computed. The SCOPE optimizer guarantees to find the lowest estimated cost plan in the search space, but this only holds for a given set of cardinality estimates and heuristics. Changing the rule configurations can impact these, thus the costs across recompilation runs with different rules are not directly comparable. A few ways in which new configurations can lead to lower costs are:

[2] If a rule does not impact the final query plan, then it makes no difference whether it is enabled or disabled. But there can also be rules that can impact the final query plan, but were missed when computing the job span. Thus, leaving these enabled can still be useful.

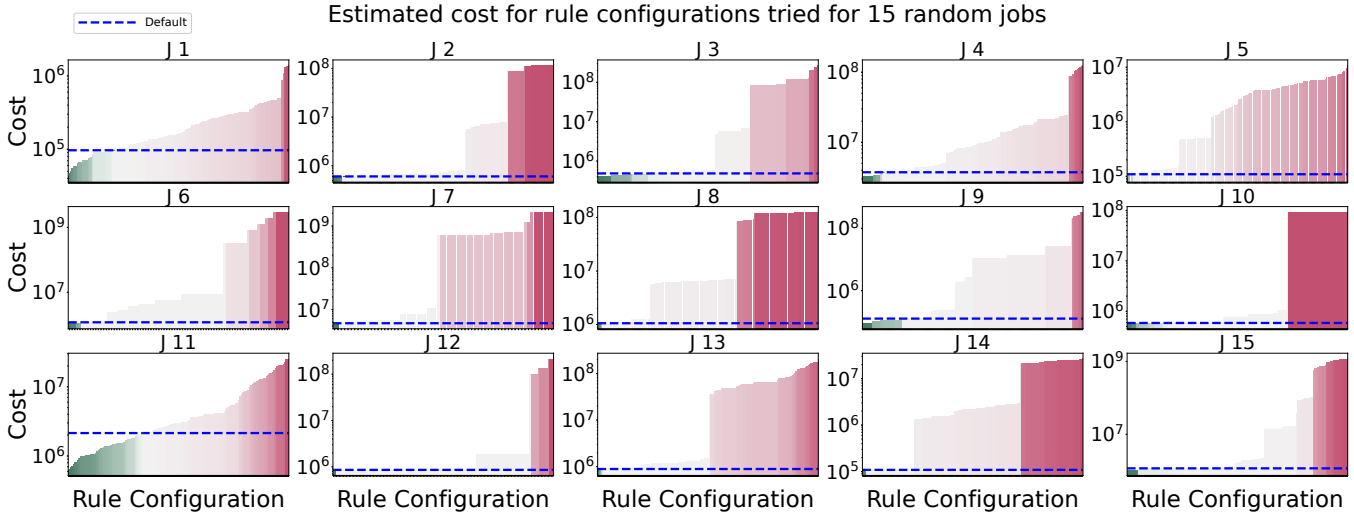


Figure 4: Comparing the estimated cost of the default rule configuration versus all the candidate rule configurations for 15 randomly selected queries in Workload A. Even though SCOPE cascade-style optimizer guarantees that the plans returned are the ones with the lowest-cost, still our approach can find plans with estimated costs lower than the default rule configuration.

(1) **Changing node properties.** Each of the nodes in the query graph have various logical and physical properties, such as cardinality estimates or costs, that are estimated based on heuristics and assumptions by SCOPE. Alternate rule configurations can change how these properties are calculated, which naturally lead to different cost estimates. For example, changing the order of filters can impact cardinality estimation (due to correlations, skew, or other reasons) at each node and hence the estimated costs.

(2) **Degree of Parallelism.** Different rules can influence the distributed nature of a query plan. This can range from how the input is distributed to different containers, to the degree of parallelism used, to the number of containers launched for a job. However, it is again not possible to explore all the distributed options exhaustively. SCOPE's search space heuristically selects a few degrees of parallelism to explore. Since the heuristics depend on the logical properties, like cardinality estimates, it is possible that with a different rule configuration, different degrees of parallelism are chosen.

Finding lower estimated cost plans with different rule configurations in SCOPE is a key difference with Bao. In PostgreSQL, the estimated costs are directly comparable under the different rule configurations considered in Bao. But, the plan found without disabling any flag would be the cheapest cost plan (but not the best plan, since mis-estimates or wrong cost model assumptions may mean that other plans can execute faster), since the current set of hints are limited to scans and joins, and they do not impact the node properties [14].

6 EXECUTING RULE CONFIGURATIONS

In the previous section, we saw how we can explore the large rule configuration space and discover interesting configurations with lower estimated costs. In this section, we dig into the runtime performance impact of these interesting rule configurations.

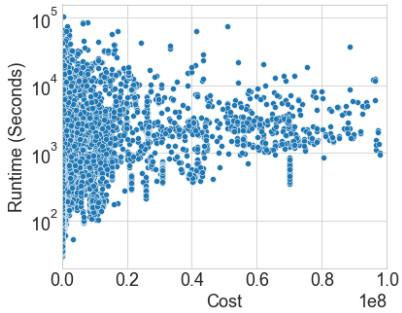


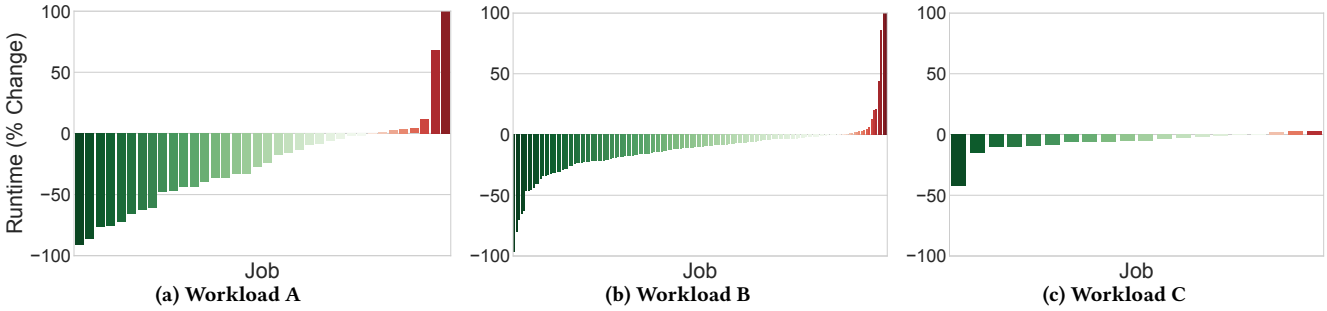
Figure 5: Comparing the costs (x-axis) v/s runtimes (y-axis). Each point is a query executed with the default rule configuration.

6.1 Choosing rule configurations to execute

One of the key components in Bao was a learned cost model which let their system choose among the different rule configurations [14]. This was possible because there were few rule configurations, and the execution time of the queries was much shorter. Thus, Bao could collect exhaustive data about different kinds of queries and rule configurations to train their model. This is not possible with hundreds of thousands of expensive jobs in production workloads and limited pre-production resources for A/B testing. Therefore, we use heuristics to choose which jobs, and which of their recompiled SCOPE plans are executed.

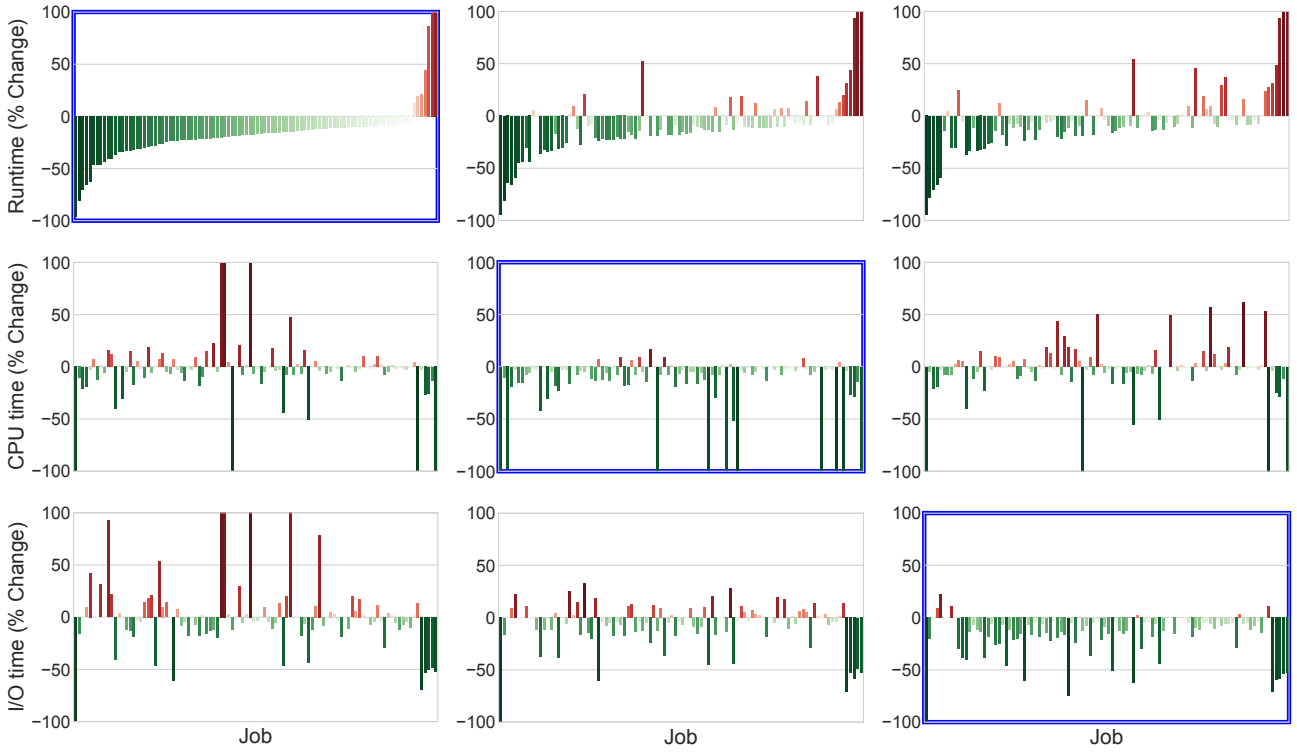
Using the cost model. Even though the estimated cost of plans recompiled with different rule configurations are not directly comparable, it is still a useful signal about the plan quality. Plans which are cheaper, or close to the cost of the default plan, indicate that the optimizer thinks the new plan is at least not a bad plan. The jobs are selected by the following heuristics:

(1) **Cheaper plans.** We sample from the jobs where the recompiled plans were clearly cheaper (difference greater than a workload specific threshold in the estimated cost) than



(a) Workload A (b) Workload B (c) Workload C

Figure 6: Jobs selected where we find cheaper plans, or jobs where the default cost is extremely low, but the runtime is not. 10 cheapest (by the cost model) rule configurations executed. Percentage change in runtime of best alternative rule configuration from default (lower is better).



(a) Rule configurations w/ best runtimes. (b) Rule configurations w/ best CPU times. (c) Rule configurations w/ best I/O times.

Figure 7: Workload B jobs. Each bar is a different query (Each plot has the same order of queries.), showing percentage change for the SCOPE metric from the default rule configuration to an alternative configuration (lower is better). For each query, we try 10 alternative rule configurations. In (a), we select configurations with best runtimes - this can lead to more regressions on other metrics, like CPU time. In (b) and (c) we select configurations with best CPU times and best I/O times, respectively.

the default plans. It makes intuitive sense to execute such plans, if only to find whether cheaper cost indicates faster runtimes.

(2) **Jobs with low cost, high runtimes.** Figure 5 shows a scatterplot of the estimated cost and runtime of the default rule configuration for all jobs during a day in Workload A. Notice, the jobs in the top left corner: these were cases with low costs, i.e., the optimizer expected them to run fast, but actually the runtimes are significantly higher. Since the cost model prediction was wrong for these jobs, it suggests that some cost model assumptions did not hold. We sample from such jobs with appropriate thresholds on cost, and runtime.

For each of the jobs selected above, we select the 10 cheapest alternative rule configurations and execute them. This lets us explore a diverse set of reasonable plans for each job. In total, we picked 2,110 rule configurations, across 211 jobs, for A/B testing in the pre-production environment.

6.2 A/B Testing Results

Executing all the alternative configurations amount to 3960 hours of execution time. Each workload has different kind of jobs, but we observe similar trends in all of them. Figure 6 shows the percentage change in runtimes from the default to the best rule configuration for the executed jobs. Notice that the maximum possible improvement in runtime can lead to a 100% change (since runtimes can not be negative), but regressions can be larger than 100%.

Runtimes. At least one of the alternative rule configurations improve runtimes for a majority of the jobs. We see a similar trend, and even magnitude of improvement in Workload A and Workload B. There are jobs that get up to 90% faster — translating to several hours saved in absolute runtimes. Workload C jobs show the same trend but the magnitude of improvements (and regressions) are smaller. This is partially because we analyzed fewer jobs in Workload C, but also because longer run times lead to lower percentage changes (e.g., improving a 10 minute job by 5 minutes will give a 50% improvement, while improving a 10 hour job by an hour is just a 10% improvement).

Different Metrics. In Section 3, we described multiple metrics of interest in SCOPE. In Figure 7, we zoom in to the Workload B results to explore how all these metrics are impacted by changing rule configurations. One natural question is: *do all the metrics improve together?* We executed ten alternative configurations for each job. We show how each metric varies for 100 jobs when we choose the best configurations for runtime (Figure 7a), CPU times (Figure 7b), or I/O times (Figure 7c). There are many cases where we see improvement on each metric (green bars from the top row to the bottom row). But more often, there is an inherent tension between all the metrics. In Figure 7a, we see regressions (red bars) in CPU time and I/O time for many jobs with improved runtimes. In Figure 7b, these regressions seem to mostly go away for the CPU times as for each job we pick one of the ten configurations that led to the best CPU time. As before, we still see regressions on I/O time, but now we also see more regressions in runtime. A similar pattern plays out in Figure 7c.

This presents a complex, but interesting optimization landscape. We could potentially have separate models that optimize for each metric individually, and choose which one to use based on the context. For instance, when there is high load on the servers, we may want to minimize one of the resource (CPU time, I/O times) while when there is less load, we may simply want the fastest runtimes. More complex policies may take into account customer requirements and so on. We could also imagine designing a loss function that seeks to balance the relative importance of each metric, and try to do the best across them all. Exploring these ideas is beyond the scope of the present work.

When the cost model is completely wrong. The cost model may be wrong, in which case, the heuristic to choose the cheapest configurations might miss good plans that are considered expensive by the cost model. To explore this scenario, we selected twenty random jobs and executed several randomly selected candidate configurations for them. We found only one example where an alternative plan was significantly better. While there is a lot of potential in learning a cost model for SCOPE, and using it to select interesting rule configurations (similar to how it was done in Bao), this experiment also shows that it is much harder to find such exceptional cases. Therefore, we decided to focus our attention on the better rule configurations we discovered by trying rule configurations based on the cost model output.

	Workload		
	A	**B**	**C**
# Queries	36	155	45
ΔRuntime	-1689s	-663s	-400s
ΔPercentage	-30%	-15%	-7%

Table 3: Average runtime change if we always choose the best known rule configurations, in terms of seconds and percentage change from the default configuration's runtime for the subset of jobs we analyzed.

Summary. Table 3 shows the runtime change, and percentage change from the default runtime, for the selected jobs if we always chose the best rule configuration (including the default one, since as we saw, for some jobs none of the different rule configurations lead to improvement). On average we see jobs get 7 – 35% faster, including jobs that speed up by 90%. These improvements would make an average runtime improvement of 400 to 1700 seconds, with many jobs improving by several hours — a significant improvement for SCOPE customers.

6.3 Why do different rule configurations improve job runtimes?

Let us dig deeper into the performance impact of changing rule configurations. The interaction of different rules in a query optimizer can be complex, therefore it is hard to pinpoint what gives rise to different emergent behaviors. We can think of these rule configurations as exploring slightly different paths within the optimizer. One hypothesis is that by disabling rules we just block certain paths — which may have been chosen due to bad estimates or heuristics in the optimizer. This may nudge the optimizer down a better path for the given job. It may have been possible for the optimizer to discover the best path by itself if its cardinality estimates and cost model assumptions were all perfect - but this is unrealistic. Nevertheless, there are a few interesting patterns we observe that lead to improved job performance.

Do lower estimated costs always result in lower runtimes? No. As described in §5, there can be many reasons for lower estimates costs, and these don't correspond to better plans. We use the low estimated cost as a signal of potentially interesting alternative plans.

Which rules change in jobs with improvements? When a new rule configuration leads to faster runtimes than the default configuration, we may want to know which changes caused the improvement. We can directly compare the enabled or disabled rules of the two configurations. But not all changes in the rule configurations are meaningful. For instance, it may be that disabling a rule has no impact on the generated query plan. Or, there may be ten differences between the two configurations, but only one of them leads to a change in query plan. We define *RuleDiff* to only look at which rule changes *actually impacted the query plans.*

Job	Runtime %change	Rules only in default plan	Rules only in best plan
Q_{A1}	−90%	JoinImpl2 SelectOnProject GroupbyBelowUnionAll ...8 more rules	CorrelatedJoinOn -UnionAll2
Q_{A2}	−86%	HashJoinImpl1 SelectOnProject SelectPredNormalized ...3 more rules	-
Q_{A3}	−75%	UnionAllToUnionAll	UnionAlltoVirtual -Dataset
Q_{B1}	−96%	TopOnRestrRemap SelectOnTrue	CollapseSelects
Q_{B2}	−80%	JoinImpl2	HashJoinImpl1
Q_{B3}	−70%	ProcesOnnUnionAll UnionAllToUnionAll	UnionAlltoVirtual -Dataset

Table 4: RuleDiff for a few sample jobs. Runtime %change is the percentage change from the default configuration's runtime to the best configuration.

Definition 6.1. **RuleDiff.** For a given job and new rule configuration, we compare the rule signature bit vectors we get using the default rule configuration, with the rule signature using the new rule configuration. The rules whose corresponding bits are equal in both the bit vectors were not directly impacted by the new rule configuration. The rules which were 1 in the default rule signature bit vector, but 0 in the new configuration's bit vector were not used in the new query plan. This could be because these rules were disabled. These rules are referred to as *rules only in default plan*. Rules only in the new plan are defined similarly.

In Table 4, we show RuleDiffs for the best configurations found for particular jobs in each workload. For Q_{A1} from Workload A- we see that one additional rule was seen in the best plan, and many additional rules were used in the default plan. The additional rule in the best plan was an off-by-default rule, which explains why it was not in the default plan.

Is enabling off-by-default rules enough for Q_{A1}? No. We have several configurations that also enable the same off-by-default rule for Q_{A1}, and disable a different subset of rules. All of these result in runtimes faster than the default, but these are $7 - 9x$ slower than the best configuration. It is not immediately obvious why this particular rule configuration leads to a plan with the fastest runtime to even experts in the SCOPE optimizer team. This highlights the fact that these rules interact in very complex ways, and it is hard to manually configure these.

Disabling rules is crucial. Some of the biggest gains occur due to disabling rules. This is clearly visible in the RuleDiffs with many rules only appearing in the default plan, but not in the best plan. In Q_{A2}, no additional rules were used in the best plan; instead six fewer rules were used, and it led to an almost 90% improvement in runtime. This supports our hypothesis that one way these rule configurations help improve performance is by blocking optimization paths that would have led to a bad plan for a given job.

Alternative rules. For both jobs Q_{A3} and Q_{B3}, rule UnionAllToVirtualDataset appears in the best plan, and UnionAllToUnionAll appears in the default plan. Both of these rules are on by default. Based on the cost model, the optimizer must have believed UnionAllToUnionAll is the better choice, but some assumptions or estimates for this job were wrong. Rather than correct these, our approach just disabled the rule that was leading to the optimizer's decision — thus guiding it towards using the other rule in the final plan. A similar motif is also observed for Q_{B2}.

6.4 Extrapolating to other jobs

In the previous section we discovered good rule configurations offline for a subset of jobs on a single day. The process of finding these configurations is expensive, particularly since it required executing plans from many alternative rule configurations for each job. For it to be useful, we need to generated these rule configurations for new jobs at *compile time*. Therefore, we seek to understand other scenarios where we can utilize the rule configurations that led to improved runtimes. Clearly they are not useful for every job in the workload. On the other extreme, these configurations seem to usually work well on jobs from the same recurring *templates*, across many days. However, as we saw in Table 1, there are tens of thousands of such templates, often with just one or a handful of jobs in each template every day. Discovering, and learning rule configurations for every template is therefore not scalable. Furthermore, even small differences in a job, such as a single different input name, will lead to different recurring template identifiers — even though the jobs may be almost identical. Therefore, we choose the rule signature as the level of granularity across which the same set of rule configurations could be useful.

Definition 6.2. **Rule signature job group.** This is the set of jobs whose default rule signature map to the same bit vector. We will refer to it as a job group.

Are the jobs with the same rule signature similar from an optimizer's perspective? A job group can have many templates, inputs, and jobs with varying runtimes. This is clearly not a homogeneous group, but it makes intuitive sense that these may have similar properties from an optimizer's perspective. The same rule signature implies that these jobs have similar operators, and went down similar paths within the query optimizer. At the same time, this may not always be true — which motivates the need for learning.

We use the results from §6.1 to select *base jobs* where alternative configurations led to improved runtimes. From the base jobs, we derive the job groups they belong to. We extrapolate that these configurations can be useful for other jobs with the same rule signature across multiple days. The shared rule signature between the base jobs, and the new unseen jobs also implies that the new configurations are not just random changes — the rules disabled or enabled in a new configuration were selected such that they impact the rules used in the base jobs.

Is learning needed? As we apply the new rule configurations to new unseen jobs, across multiple weeks, we observe two scenarios:

- **The new configuration does not cause regressions.** There are cases where a configuration that led to runtime improvements in a base job appears to mostly work well on all the jobs in a job group of the given workload. An example from Workload A was shown in Section 1, Figure 1. These rule configurations appear to benefit almost all jobs sampled randomly across almost four weeks in the same job group. This is an ideal situation, and we could derive the benefit of our approach without much risk of a regression. Clearly, this behaviour could change in the future as the predicates and input streams of these jobs may evolve. This risk can be mitigated by re-running our pipeline every week.
- **New configuration can cause a regression.** A more common scenario is where a configuration leads to improvements for some jobs, but a regression for others in the same job group. In such situations, we try to use a learned model to choose between the configurations based on the input features of the new job. This is explored further in §7.

What fraction of the daily jobs may be impacted? We did our analysis above on a fraction of randomly chosen jobs from a single day. Let us do some back of the envelope calculations for Workload B jobs to estimate the impact these rule configurations may have on the whole workload. It is not very critical to improve performance on the short running jobs since they consume significantly less resources. There are about 10%, or 1624 jobs daily that last longer than 5 minutes and these map to 300 different job groups. We uniformly sampled about half of these, and ran our recompilation pipeline for 838 jobs. From these, we used heuristics to select 155 jobs (or about 20%) for execution of the 10 cheapest alternative plans. We find runtime improvements in 120 jobs, with the change ranging from -3% to -90%. These 120 jobs belong to 44 job groups. This suggests that at least $10-20\%$ of the long running daily jobs could be improved using this pipeline. However, the improvements described in this section relied on executing multiple plans. In a real system, we will need to choose one rule configuration each time. Next, we formulate this as a learning problem and do some very preliminary analysis to see how well can we choose the best rule configurations.

7 LEARNING RULE CONFIGURATIONS

In this section, we see how to learn to choose a rule configuration given a new job.

7.1 Formulating the learning problem

Dataset. For training a learned model, we collect a dataset of runtimes for a given combination of a job and a rule configuration. We select three rule signatures from Workload B, and collect data on queries over a few weeks whose default rule signature maps to these (i.e., jobs that belong in the same job group). All of these job groups have more than a dozen jobs every day, and there is no one rule configuration that always leads to improvements. For each job group, we run our pipeline for one to three jobs, and find configurations leading to improvements. We select the three fastest (runtime) configurations of each job — therefore, we get up to K candidate configurations for each job group. Then, we sample M jobs from all the jobs mapping to these job groups during a period of two weeks, and execute each of the K configurations for every job.

Learning Problem. We treat the dataset of samples in each job group as an independent learning problem. The objective is to select one of the K candidate configurations for a given query. We use supervised learning to train a model to pick of of the K configurations for each job group.

7.2 Featurization

For any learning based approach, we need to featurize each job. Since only the decisions related to join ordering were considered in Bao, the PostgreSQL query plans were relatively small trees. In contrast, SCOPE query plans are large DAGs, consisting of 100*s* to even 1000*s* of operator nodes, including widespread use of customer user defined operators. Furthermore, a SCOPE script is compiled into an optimized query plan, which is converted into a DAG of stages and is executed in a distributed fashion, and altogether the SCOPE engine emits several pieces of disparate logs from each of these steps that are hard to featurize, and may not be even relevant for the purpose of a given task. Therefore, featurizing a SCOPE job is a challenging problem.

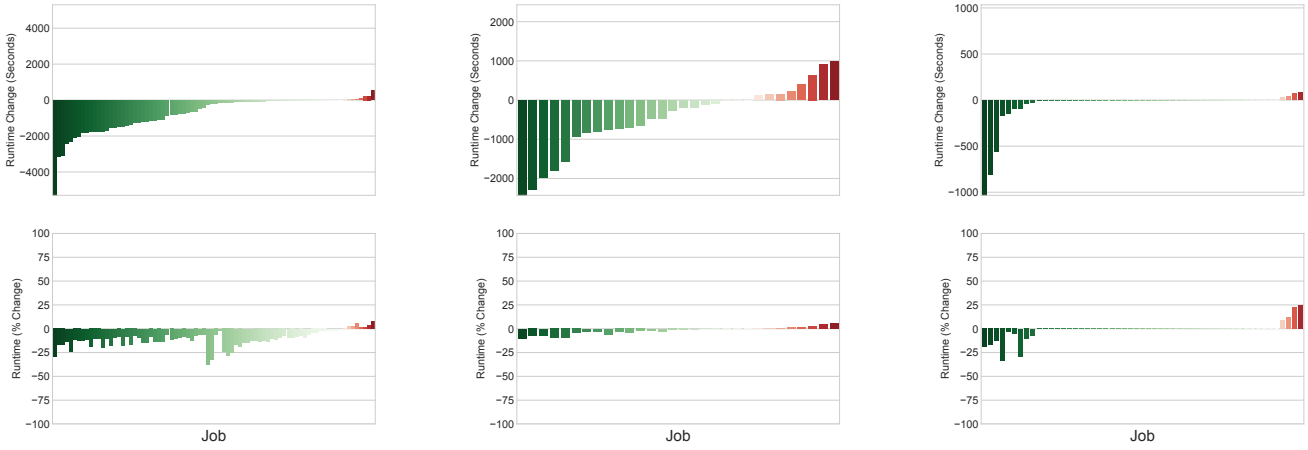
Feature vector. We use a feature vector to capture the most important features for choosing between the K rule configurations for samples from a particular job group. These include: (1) Job level features. These are global properties of the particular job which includes estimated input size, hash of the inputs, and the hash of the query template. (2) Rule configuration features. For each configuration we use the estimated cost of a plan and a bit vector representing the RuleDiff with the default configuration. (3) Query graph features. We reserve a spot in the feature vector for each possible operator type. These include operator id, and average estimated cost and cardinality for all repetitions of the operator.

Encoding features. We encode all continuous features using min-max normalization to scale its value in between 0 and 1. For categorical features with small alphabet sizes we use one hot encoding. For categorical features with large alphabet sizes, we use a deterministic hashing scheme encoding each value to one of 50 bins [21].

7.3 Learning Model

Lightweight model. For each job group, we use a fully connected neural network with one hidden layer of size 1024. These models have a size of about 30MB each and take a minute to train.

Loss function. We treat the learning as a regression problem where the objective is to estimate each of the normalized K runtimes corresponding to the K possible rule configurations for a query. A typical way to optimize regression tasks is to use mean squared error. But, this tries to get a precise estimate for each candidate configuration. This may often not be needed because we really only care about choosing the fastest configuration. Instead, we use a continuous version of the cross entropy loss, referred to as the binary cross entropy loss in PyTorch [20].



(a) Job Group 1 (b) Job Group 2 (c) Job Group 3

Figure 8: Performance of the learned model relative to the default rule configuration. Change in runtimes (above), and percentage change in runtimes (below) from the default lower is better. These are unseen queries from three job groups sampled over weeks. For each query, the learned neural net model chooses from K potential rule configurations (including default).

	1			2			3		
	Mean	90P	99P	Mean	90P	99P	Mean	90P	99P
Best	5458	14K	14.8K	19.8K	26K	27K	2966	13.8K	15.3K
Default	6461	16.3K	18.3K	20.7K	26.9K	28.9K	3304	14.7K	16.8K
Learned	**5724**	**14.7K**	**15.4K**	**20.2K**	**26.2K**	**27K**	**3252**	**14.6K**	16.8K

Table 5: Runtimes (seconds) for job groups 1, 2 and 3 with the best (known), default and learned configurations.

7.4 Learning Results

Setup. Job group 1 has 201 jobs and 10 possible rule configurations, job group 2 has 75 jobs and 7 possible rule configurations, and job group 3 has 157 jobs and 10 possible rule configurations. We randomly split the jobs in all three job groups into 20% validation set, and 40% training and test sets. We tune the hyperparameters of the model on the validation set, and report results on the test set.

Results summary. The runtimes are presented in Table 5, and the changes from the default runtime for each query are shown in Figure 8. Overall, we see improvements in each job group, but there are always some regressions as well.

Job group 1. There are large improvements, up to an hour faster on runtimes, across a large fraction of the jobs. Notice that the runtimes have different scales, thus the magnitude of percentage improvements do not align with the absolute runtime improvements (shorter running jobs can have larger improvements). As can be seen in Table 5, the learned model is close to making the best decisions.

Job group 2. These are significantly longer running jobs — one impact is that the percentage values appear to be relatively smaller than other job groups. But we see consistent improvements of up to 2000 seconds across multiple jobs, while the regressions appear to be smaller.

Job group 3. This appears to be the hardest job group to optimize. Notice, the jobs without green or red bars — these are cases where our learned model recommends the default configuration. But it does find improvements of up to a 1000 seconds for multiple other jobs. Meanwhile, the regressions have much lower magnitudes. But, there is potential for more significant improvements, as we can see from the best runtimes in Table 5.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented how the SCOPE optimizer — an industry-grade cloud-enabled query optimizer — can take advantage of a learning approach, such as the one introduced in Bao [14]. We introduced the ideas of a rule signature and job span that help us navigate the large space of rule configurations, efficiently discover interesting configurations, and even extrapolate them to other unseen jobs. Our results over three production workloads show that we can achieve up to 90% better runtime latencies, with 7 – 35% improvement on average, while requiring minimal changes to the SCOPE optimizer. We believe that the directions outlined in this paper open many doors, some of which we are currently exploring ourselves. For instance, there are multiple ways to improve the heuristics used for the generation of the job span and the candidate rule configurations. Such improvements can discover independent subsets of rules, which will make the space of rule configurations smaller, therefore enabling exploration of better configurations. We also plan to use feedback from the execution results to guide future iterations of the configuration search. Finally, there are also hundreds of additional configurable options in the SCOPE optimizer, and we may be able to generate interesting behaviours by also including them in the candidate configurations.

REFERENCES

[1] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-Optimizing Data-Parallel Computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (San Jose, CA) (NSDI'12). USENIX Association, USA, 21.

[2] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In Proceedings of the 2018 International Conference on Management of Data. 221–230.

[3] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. Proceedings of the VLDB Endowment 1, 2 (2008), 1265–1276.

[4] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In Proceedings of the 2016 International Conference on Management of Data. 215–226.

[5] Anshuman Dutt, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2020. Efficiently Approximating Selectivity Functions using Low Overhead Regression Models. Proc. VLDB Endow. 13, 11 (2020), 2215–2228. http://www.vldb.org/pvldb/vol13/p2215-dutt.pdf

[6] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. Proceedings of the VLDB Endowment 12, 9 (2019), 1044–1057.

[7] Goetz Graefe. 1995. The cascades framework for query optimization. IEEE Data Eng. Bull. 18, 3 (1995), 19–29.

[8] Jayant R Haritsa. 2010. The Picasso database query optimizer visualizer. Proceedings of the VLDB Endowment 3, 1-2 (2010), 1517–1520.

[9] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In Proceedings of the ACM Symposium on Cloud Computing. 416–427.

[10] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at microsoft. In Proceedings of the 2018 International Conference on Management of Data. 191–203.

[11] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. arXiv preprint arXiv:1809.00677 (2018).

[12] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. arXiv preprint arXiv:1808.03196 (2018).

[13] 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.

[14] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2020. Bao: Learning to Steer Query Optimizers. arXiv preprint arXiv:2004.03814 (2020).

[15] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. arXiv preprint arXiv:1904.03711 (2019).

[16] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 1–4.

[17] Corp. Microsoft. 2020. SQL Server. Retrieved November 23, 2020 from https://www.microsoft.com/en-us/sql-server/

[18] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters. In 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW). IEEE, 154–157.

[19] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. arXiv preprint arXiv:1905.06425 (2019).

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[21] Dipanjan (DJ) Sarkar. 2019. Categorical Data. https://towardsdatascience.com/understanding-feature-engineering-part-2-categorical-data-f54324193e63

[22] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. AutoToken: predicting peak parallelism for big data analytics at Microsoft. Proceedings of the VLDB Endowment 13, 12 (2020), 3326–3339.

[23] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. 2013. F1: A distributed SQL database that scales. (2013).

[24] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 99–113.

[25] Florian M Waas. 2008. Beyond conventional data warehousing—massively parallel data processing with greenplum database. In International Workshop on Business Intelligence for the Real-Time Enterprise. Springer, 89–96.

[26] Florian Wolf, Michael Brendle, Norman May, Paul R Willems, Kai-Uwe Sattler, and Michael Grossniklaus. 2018. Robustness metrics for relational query execution plans. Proceedings of the VLDB Endowment 11, 11 (2018), 1360–1372.

[27] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 1–8.

[28] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi