SAMPLING-BASED QUERY EXECUTION TIME PREDICTION AND QUERY RE-OPTIMIZATION

by

Wentao Wu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 05/13/15

The dissertation is approved by the following members of the Final Oral Committee:

Jeffrey F. Naughton, Professor, Computer Sciences
David J. DeWitt, Emeritus Professor, Computer Sciences
AnHai Doan, Professor, Computer Sciences
Jignesh M. Patel, Professor, Computer Sciences
C. David Page Jr., Professor, Biostatistics and Medical Informatics

To my parents, Huixian Cheng and Chun Wu.

Acknowledgments

I am forever indebted to my advisor, Prof. Jeffrey Naughton, for his kindness, patience, and endless help, not only in my research, but also in my life and career. The fundamental idea of this dissertation originated from a 30-minute discussion with Jeff in a snowing, winter morning in 2011, after I had tried some other frustrating approaches for several months. At that time, I was excited by the idea but I was really not sure how far I could go from there. It is Jeff's guidance, encouragement, and support that help me eventually arrive at the end of this long journey. I will never forget the extremely insightful, detailed comments from Jeff regarding everything including ideas, methods, paper writing, and talk presentation. The inspiration I gain during this training and the skills I learnt from him is invaluable to my future career. I owe my deepest gratitude to him.

I am fortunate to have Prof. David DeWitt, Prof. AnHai Doan, and Prof. Jignesh Patel from the Database Group on my committee. I happened to meet David relatively late in my Ph.D study, but I had been inspired by his work and passion in database system research along the way. I am grateful to him for his time and feedback on this dissertation. AnHai and Jignesh raised lots of tough but really insightful questions on the work presented here, which deepened my understanding on both the problems and the solutions. They also offered numerous valuable comments and suggestions that help improve this dissertation. I am thankful to them for all their contributions. I am also thankful to Prof. Christopher Ré for his early input on this work when he was at the University of Wisconsin-Madison.

I thank Prof. David Page for teaching me machine learning and generously agreeing to serve on my committee. I thank Prof. Remzi Arpaci-Dusseau, Prof. Eric

Bach, Prof. Shan Lu, Prof. Mary Vernon, Prof. Tonghai Yang, and Prof. Chunming Zhang. This dissertation benefits from what I learnt from their classes. I also want to thank Prof. Jin-Yi Cai for helpful discussions. Special thanks to Angela Thorp for helping me through various administration stuff.

Over the years, I took several internships at industry companies. I spent two fruitful summers at NEC Labs. This dissertation indeed started from my intern project there. I thank all my mentors: Yun Chi, Hakan Hacıgümüş, Junichi Tatemura, and Shenghuo Zhu, for their great help and guidance during my internships. I also spent one wonderful summer at Google, working with the F1 team on query optimization. Special thanks go to my mentors: John Cieslewicz, Stephan Ellner, Felix Weigel, and Chad Whipkey. The experience I gained from building real large systems is precious to my future career.

I want to thank all past and current students in the Database Group. It was great honor for me to be admitted into this great group, with so many smart, friendly, and hard-working fellow students. Throughout the years, I made a lot of friends and learnt lots of things from them. In some sense, I could not have finished this dissertation without their help. Thanks also go to many friends in the Computer Sciences Department outside the Database Group, and go to other friends in this great university and this great city. I apologize that I could not make a complete list here that would otherwise make this paragraph ridiculously long.

Finally, I would like to thank my parents for their endless love and support in my life. My mother started teaching me the importance of being a knowledgeable and well-educated person when I was still a kid. It was her hope that I could pursue a Ph.D degree. I wish she could still see this dissertation, and I believe she would be very happy. My father always gives me the freedom to choose whatever I want to do. The only decision he made for me during my life, as far as I can remember, was to ask me to give up the offer from the Chemistry Engineering Department of a top Chinese university and pursue a computer science degree, although he knew little about computer science at that time. Without this crucial decision, I would not have the chance to work in the exciting area of database systems and write this dissertation. I therefore dedicate this dissertation to them.

Contents

List of Tables

List of Figures

Abstract

The problem of query execution time prediction and query optimization is fundamental in database systems. Although decades of research has been devoted to this area and significant progress has been made, it remains a challenging problem for many queries in real-world database workloads.

In this dissertation, we study a simple idea based on calibration of existing cost models used by current query optimizers. The calibration procedure mainly relies on refining cardinality estimates via sampling. We show how sampling can be effectively employed to provide better query execution time predictions for both single-query and multi-query workloads. Although our approaches outperform the state of the art, they are still not perfect. We therefore turn to the complementary problem of quantifying uncertainty in query execution time prediction. Uncertainty information could be useful in many applications such as query optimization, query scheduling, and query progress monitoring. Specifically, we develop a predictor that can provide distribution rather than point estimates for query execution times. Again, sampling plays an important role in the development of this predictor. We show that the variances of the estimated distributions are strongly correlated with the actual prediction errors. While this line of work improves query execution time estimates, it cannot make a query run faster. Given that the quality of cardinality estimates is crucial to cost-based query optimization, it is natural to ask if the refined cardinality estimates via sampling could also be useful to query optimizers. We further study this problem and propose an iterative sampling-based compile-time query re-optimization procedure. We show the efficiency and effectiveness of this approach both theoretically and experimentally.

Chapter 1

Introduction

Cloud computing and big-data analytics are two hot spots intriguing to the database research community today. Providing cloud-based database services, known as "database as a service" (DaaS), has been a focus of many big-data companies such as Amazon, Google, and Microsoft. DaaS providers, however, face challenges that are both familiar and unfamiliar to traditional database researchers. On one hand, classic database management issues remain. For example, query optimization becomes even more difficult as data volume and query complexity keep growing in big-data applications. On the other hand, new database management issues arise. For instance, admission control, query scheduling, and resource planning, which were previously duties of database administrators, now become critical issues in the context of DaaS.

In this dissertation, we start by studying a fundamental problem underlying these database management challenges in DaaS: predict the execution time of a query. A DaaS provider has to manage infrastructure costs as well as honor service level agreements (SLAs), and many system management decisions can benefit from prediction of query execution time, including:

- *Admission Control*: Knowing the execution time of an incoming query can enable cost-based decisions on admission control [88, 96].

- *Query Scheduling*: Knowing query execution time is crucial in deadline and latency aware scheduling [27, 42].
- *Progress Monitoring*: Knowing the execution time of an incoming query can help avoid "rogue queries" that are submitted in error and take an unreasonably long time to execute [67].
- *System Sizing*: Knowing query execution time as a function of hardware resources can help in system sizing [91].

In a sense, this is a very old problem that has perplexed database practitioners for decades: query optimizers rely on reasonable cost models for query execution time estimates. However, existing cost models are insufficient for emerging DaaS applications. They are designed for query optimization purpose, which is somehow robust to even significant errors presented in cost estimates. In previous work, it has been shown that just relying on existing cost models can lead to orders-of-magnitude errors in query execution time estimates [11, 36]. Researchers then turned to machine-learning based approaches, and there has been an intensive line of recent work in this direction (e.g., [11, 33, 36]). In this dissertation, we study this problem from a different perspective: we can obtain better predictions via a systematic yet lightweight *calibration* procedure of existing cost models. This approach shows competitive and often much better execution time estimates compared with previous approaches. We then extend this approach to predict query execution times for multiple, concurrently-running queries. We further study two applications of this framework: (1) quantifying uncertainty in query execution time estimation and (2) improving query plans returned by the query optimizer.

1.1 Prediction For Single-Query Workloads

Accurate query execution time estimation is not a trivial problem. While the details of the cost models used by current query optimizers differ, they almost follow the same implementation logic: estimate (1) the amount of "work" that a query

needs to accomplish (e.g., the number of CPU instructions executed, the number of disk pages accessed, etc.) and (2) the time spent per unit work (referred to as "cost unit" in the following). Multiplying these two quantities gives the estimated execution time. The challenge here is to get accurate values for both quantities. Unfortunately, neither of them is easy to estimate: the former requires accurate selectivity/cardinality estimation, a well-known difficult problem in the literature of database and statistics research; the latter requires accurate knowledge about system execution status, such as CPU or I/O speeds at a given instant, which appears to fluctuate due to concurrent workloads. In practice, optimizers therefore have to adopt (often invalid) assumptions and heuristics to estimate these quantities.

Our approach to this problem is a systematic calibration procedure of the cost model [94]. We calibrate the cost units by using a set of primitive queries (basically sequential scans and index-based scans), and we calibrate (or refine) the selectivity estimates by using a sampling-based approach. While the idea of sampling-based selectivity estimation goes back two decades [50, 62], it is not used in current query optimizers mainly due to efficiency reasons, because sampling has to be used for all (perhaps hundreds or even thousands of) plans explored by the optimizer. Our key observation here is that, for the purpose of execution time prediction, we only need to use sampling *once* (for the final optimal plan chosen by the optimizer). We demonstrate that this idea can lead to better predictions than those reported in previous work. We present the details of this work in Chapter 2.

1.2 Prediction For Multi-Query Workloads

In practice, it is rarely the case that there is only one single query running in a dedicated database system. Rather, in a typical DaaS scenario, multiple queries are usually concurrently running, and workloads are subject to change from time to time. There is only a fraction of previous work aimed at query execution time prediction for multi-query workloads [10, 33]. However, this work assumes that the workloads are static, namely, all possible queries in the workloads are known beforehand. They then develop prediction techniques based on machine learning.

We go one step further by developing a prediction framework that is not restricted to static workloads [93]. Specifically, we extend our framework of predicting query execution time for single-query workloads to multi-query workloads, based on the following observation. Note that, in this scenario, the amount of work a query needs to accomplish is the same as that if the query were running without interactions from other queries. The existence of other queries, however, does affect the values of the cost units. For example, two concurrently-running I/O-intensive queries might slow down each other. Built on top of this observation, we develop a combination queueing model and buffer pool model to estimate cost units under concurrent database workloads. We present the details of this work in Chapter 3.

1.3 Measuring Uncertainty

Although we have made progress in predicting execution times for both standalone and concurrently-running queries, we find that sometimes significant prediction errors could still occur. While continuing improving the prediction accuracy is still a promising way to go, we turn to another orthogonal but complementary problem of quantifying the *uncertainty* in the prediction [95]. It is a general principle that predictions are more convincing and useful if they are accompanied by confidence intervals. Uncertainty information is also useful in a couple of database applications including query optimization, query scheduling, and query progress monitoring.

Our basic idea is to view the two aforementioned quantities (i.e., amount of work and cost units) as random variables rather than fixed constants. We then again use sampling-based methods to estimate a distribution of likely query running times. Recent work has shown the effectiveness of leveraging distribution information in query scheduling [26]. We present the details of this work in Chapter 4.

1.4 Improving Query Plans by Re-Optimization

So far, we have systematically studied the problem of estimating query execution time and its uncertainty. In a nutshell, our idea is based on the fact that sampling can

provide us with better selectivity estimates than histogram-based approaches used by current optimizers, especially on correlated data. However, sampling incurs additional overhead so it should be used conservatively. Recall that, in our previous work, we used sampling as a "postprocessing" step once per query to "validate" selectivity estimates, or in other words, detect potential errors in optimizer's cost estimates, for the final plan returned by the optimizer. But, if there were really significant errors, the optimality of this final plan would itself be questionable. A natural question is then if sampling could be further used to improve query plans.

We explore this question in Chapter 5. Our basic idea is simple: if significant cardinality estimation errors are detected, we go one step further to let the optimizer re-optimize the query by also feeding it the cardinality estimates refined via sampling. This gives the optimizer second chance to generate a different, perhaps better, plan. Note that we can again apply the sampling-based validation step to this new plan returned by the optimizer. It therefore leads to an iterative procedure based on feedback from sampling: we can repeat this optimization-then-validation loop until the plan chosen by the optimizer does not change.

We further study this re-optimization procedure in detail, both theoretically and experimentally. Our theoretical analysis suggests that the efficiency of this procedure and the quality of the final plan returned can be guaranteed under certain assumptions, and our experimental evaluation on the TPC-H benchmark database [6] as well as our own database with highly correlated data demonstrates the effectiveness of this approach.

Chapter 2

Query Execution Time Prediction for Single-Query Workloads

Predicting query execution time is useful in many database management issues including admission control, query scheduling, progress monitoring, and system sizing. Recently the research community has been exploring the use of statistical machine learning approaches to build predictive models for this task. An implicit assumption behind this work is that the cost models used by query optimizers are insufficient for query execution time prediction.

In this chapter we challenge this assumption and show while the simple approach of scaling the optimizer's estimated cost indeed fails, a properly calibrated optimizer cost model is surprisingly effective. However, even a well-tuned optimizer cost model will fail in the presence of errors in cardinality estimates. Accordingly we investigate the novel idea of spending extra resources to refine estimates for the query plan after it has been chosen by the optimizer but before execution. In our experiments we find that a well calibrated query optimizer model along with cardinality estimation refinement provides a low overhead way to provide estimates that are always competitive and often much better than the best reported numbers from the machine learning approaches.

2.1 Introduction

Predicting query execution time has always been desirable if somewhat elusive capability for database management systems. This capability has received a flurry of attention recently, perhaps because it has become increasingly important in the context of offering databases as a service (DaaS).

Recent work on predicting query execution time [11, 36, 88, 96] has focused on various machine learning techniques, which treat the database system as a black box and try to learn a query running time prediction model. This move toward black box machine learning techniques is implicitly and sometimes explicitly motivated by a belief that query optimizers' cost estimations are not good enough for running time prediction. For example, in [36], the authors found that using linear regression to map the cost from Neoview's commercial query optimizer to the actual running time was not effective (see Figure 17 of [36]). In [11], the same approach was used to map PostgreSQL's estimate to the actual execution time, and similar disappointing results were obtained (see Figure 5 of [11]).

It is clear from this previous work that post-processing the optimizer cost estimate is not effective. However, we argue in this chapter that this does not imply that optimizer estimates are not useful — to the contrary, our experiments show that if the optimizer's internal cost model is tuned before making the estimate, the optimizer's estimates are competitive with and often superior to those obtained by more complex approaches. In more detail, for specificity consider the cost model used by the PostgreSQL query optimizer:

Example 2.1 (PostgreSQL's Cost Model)**.** *PostgreSQL's optimizer uses five parameters (referred to as* cost units*) in its cost model:* $\mathbf{c} = (c_s, c_r, c_t, c_i, c_o)^\intercal$*, defined as follows:*

1) c_s*:* seq_page_cost*, the I/O cost to sequentially access a page.*

2) c_r*:* random_page_cost*, the I/O cost to randomly access a page.*

3) c_t: cpu_tuple_cost*, the CPU cost to process a tuple.*

4) c_i: cpu_index_tuple_cost*, the CPU cost to process a tuple via index access.*

5) c_o*:* cpu_operator_cost*, the CPU cost to perform an operation such as hash.*

The cost C_O *of an operator* O *in a query plan is then computed by a linear combination of* c_s, c_r, c_t, c_i, *and* c_o*:*

$$C_O = \mathbf{n}^\intercal \mathbf{c} = n_s \cdot c_s + n_r \cdot c_r + n_t \cdot c_t + n_i \cdot c_i + n_o \cdot c_o. \tag{2.1}$$

The values $\mathbf{n} = (n_s, n_r, n_t, n_i, n_o)^\intercal$ *here represent* the number of pages sequentially scanned, the number of pages randomly accessed*, and so forth, during the execution of the operator* O*. The total estimated cost of a query plan is then simply the sum of the costs of the individual operators in the query plan.*

The accuracy of C_O hence depends on both the accuracy of the c's and the n's. In PostgreSQL, by default, $c_s = 1.0$, $c_r = 4.0$, $c_t = 0.01$, $c_i = 0.005$, and $c_o = 0.0025$. The cost C_O in Equation (2.1) is thus reported in units of sequential page I/O cost (since $c_s = 1.0$). Note that these cost units were somewhat arbitrarily set by the optimizer designers with no knowledge of the system on which the query is actually being run. Using linear regression to map an estimate so obtained will only work if the ratios among these units are correct, and not surprisingly, these default ratios were far from correct on our systems.

Of course, the accuracy of C_O also depends on the quantities n_s, n_r, n_t, n_i, and n_o. Determining accurate values for these n's is not a matter of calibration — rather, it is a matter of good cardinality estimation in the optimizer. Hence one could say that we have reduced the problem of query time prediction to the previously unsolved problem of cardinality estimation. In a sense this is true, but further reflection reveals that we are solving a subtly but significantly different problem.

In their traditional role, cardinality estimates are required for every cardinality encountered as the optimizer searches thousands or tens of thousands of alternative plans. This of course means that the estimation process itself must be extremely efficient, or long optimization times will result. But our problem is different: we must determine cardinalities for the single plan that the optimizer has already chosen. The fact that we are working on a single plan means we can afford to spend

some extra time to improve the original optimizer estimates. Specifically, in this chapter we consider using sampling-based approaches to refine these estimates. We believe that although sampling-based approaches may be too expensive to be used while *searching* for good query plans, they can be practically used for *correcting* the erroneous cardinality estimates in a ready-to-be-executed query plan.

Our experiments show that if we correctly calibrate the constants in the optimizer's cost model, it yields good query execution time estimates when the cardinality estimates are good (as is the case in, for example, the uniformly distributed data set variants of the TPC-H benchmark). Furthermore, "expensive" techniques such as sampling can be effectively used to improve the cardinality estimates for the chosen plan. Putting the two together yields cost estimates that are as good or better than those obtained by previously studied machine learning approaches.

The rest of this chapter is organized as follows. We first give an overview of our cost-model based approach in Section 2.2. We then discuss the two error-correction steps, i.e., calibrating cost units and refining cardinality estimates, in Sections 2.3 and 2.4, respectively. We further conduct extensive experimental evaluations and present our results in Section 2.5. We discuss related work in Section 2.6 and summarize this chapter in Section 2.7.

2.2 The Framework

Example 2.1 demonstrates that errors in $\mathbf{c}$ and $\mathbf{n}$ might prevent us from leveraging $C_O = \mathbf{n}^T\mathbf{c}$ to predict query execution time. Our basic idea is simply to correct these errors in a principled fashion.

As illustrated in Figure 2.1, our framework consists of two error-correction stages, namely, an *offline* profiling stage to calibrate the $\mathbf{c}$, and an *online* sampling stage to refine the $\mathbf{n}$:

- *Offline profiling to calibrate* $\mathbf{c}$:

  The errors in $\mathbf{c}$ reflect an inaccurate view of the underlying hardware and database system. To correct this, instead of using the default values assigned

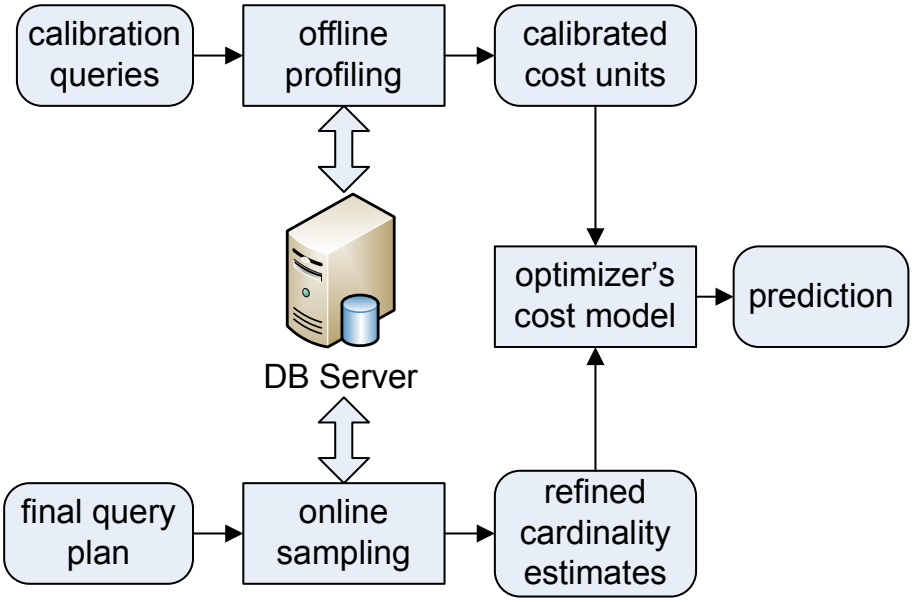


Figure 2.1: The architecture of our framework.

by the query optimizer, we calibrate them by running a family of profiling queries on the system on which the queries will be run. Note that this profiling stage is offline. Moreover, it only needs to be run once as long as the underlying hardware and software configuration does not change.

We describe the details of the profiling stage in Section 2.3.

- *Online sampling to refine* **n**:

  The errors in **n** reflect errors in cardinality estimation. Once the query plan has been chosen by the query optimizer, we re-estimate cardinalities, if necessary, using a sampling-based approach. Although using sampling for cardinality estimation is well-known, current DBMS optimizers exclude sampling from their implementations, perhaps due to the additional overhead sampling incurs. However, since we only have to estimate cardinalities for one plan, the overhead of sampling is affordable in practice. We describe the details of the sampling stage in Section 2.4. We note that the important idea is that there is an opportunity to spend extra time refining cardinality estimates once a

query plan has been chosen; sampling is one example of how that could be done, and finding other techniques is an interesting area for future work.

The advantages of our framework for predicting query execution time include:

- *Lightweight*: The profiling step is fast and so can be conducted in a new hardware environment quickly. The sampling step, as will be shown, introduces small (usually $< 10\%$) and tunable overhead.
- *No training data needed*: Unlike machine-learning based approaches, which rely on training data representative of the actual workload, our framework does not rely on such a training data set and so can handle *ad hoc* queries well.
- *White-box approach*: Instead of sophisticated statistical models (e.g., SVM and KCCA), which are often difficult to understand for non-experts, our framework adopts an intuitive white-box approach that fits naturally into the existing paradigm of query optimization and evaluation in relational database systems.

2.3 Calibrating Cost Units

In this section we consider the task of calibrating the cost units in the optimizer cost model to match the true performance of the hardware and software on which the query will be run. It turns out that closely related problems have been studied in the context of heterogeneous DBMS [32], DB resource virtualization [84], and storage type selection [98]. Previous work, however, has focused either on cost models for particular operators (e.g., selections and 2-way joins in [32]), or on a subset of cost units dedicated to a particular subsystem of the DBMS (e.g., CPU in [84] and I/O in [98]). We build on this previous work following their technique of using a set of *calibration queries*. The basic idea is that for each cost unit to be calibrated, one designs some particular query that isolates this parameter from the others. In practice, this is not completely straightforward in that not every cost unit can be isolated in a single query.

2.3.1 Guiding Principles

Ideally, we wish to have one calibration query for each parameter. However, this is not always possible. For instance, there is no SQL query which involves the *cpu_operator_cost* but not the *cpu_tuple_cost*. A natural generalization is then to use k calibration queries for k parameters, as was done in [84]. The following example illustrates this idea.

Example 2.2 (Calibration Query)**.** *Suppose* `R` *is some relation that is buffer pool resident. We can use the following two calibration queries to obtain the parameters* cpu_tuple_cost *and* cpu_operator_cost*:*

- q_1*:* `SELECT * FROM R`
- q_2*:* `SELECT COUNT(*) FROM R`

Since `R` *is memory resident, there is no I/O cost for* q_1 *or* q_2*.* q_1 *only involves the parameter* cpu_tuple_cost*, while* q_2 *involves both* cpu_tuple_cost *and* cpu_operator_cost *(due to the* `COUNT` *aggregation). Suppose the execution time of* q_1 *and* q_2 *are* t_1 *and* t_2*, respectively. Since the overhead due to* cpu_tuple_cost *for* q_1 *and* q_2 *are the same, we can then infer* cpu_operator_cost *with respect to the execution time* $t_2 - t_1$*.*

Specifically, let the number of tuples processed be n_t*, and the number of CPU operations be* n_o*, as in Equation (2.1). In PostgreSQL's cost model, a CPU operation means things like* adding two integers, hashing an attribute, *and so on.* n_o *is thus the number of such operations performed. On the other hand,* n_t *is the number of input* tuples *(sometimes the output tuples or the sum of both, depending on the specific operator). Here, for* q_1*, the cost model will only charge one* c_t *per tuple, since the CPU merely reads in the tuple without any further processing. For* q_2*, in addition to charging one* c_t *per tuple for reading it, the cost model will also charge one* c_o *per tuple for doing the aggregation (i.e.,* `COUNT`*), and hence the total CPU cost is estimated to be* $n_t c_t + n_o c_o$*. Note that for this particular query* q_2*, we coincidentally have* $n_t = n_o = |R|$*. In general,* n_t *and* n_o *could be different. For example, for the* sort *operator,* n_t *is the number of input tuples, and* n_o *is the number of comparisons made. In this case,* $n_o = n_t \log n_t$*.*

Now suppose that T_t *is the time for the CPU to process one tuple, and* T_o *is the time for one CPU operation. We then have*

$$\begin{aligned} t_1 &= T_t \cdot n_t, \\ t_2 &= T_t \cdot n_t + T_o \cdot n_o. \end{aligned}$$

Solving these two equations gives us the values of T_t *and* T_o*, in turn determines* c_t *and* c_o *in Equation (2.1).*

In general, with a set of calibration queries Q, we first estimate $\mathbf{n}_i$ for each $q_i \in Q$ and then measure its execution time t_i. With $\mathbf{N} = (\mathbf{n}_1, \cdots, \mathbf{n}_k)^\intercal$ and $\mathbf{t} = (t_1, \cdots, t_k)^\intercal$, we can solve the following equation for $\mathbf{c}$:

$$\mathbf{Nc} = \mathbf{t},$$

which is just a system of k equations.

The next question is then, given a set of optimizer cost units $\mathbf{c}$, how to design a set of calibration queries Q. Our goal is to design a set with the following properties:

- *Completeness*: Each cost unit in $\mathbf{c}$ should be covered by at least one calibration query $q \in Q$.
- *Conciseness*: Each query $q \in Q$ should be necessary to guarantee completeness. In other words, any subset $Q' \subset Q$ is not complete.
- *Simplicity*: Each query $q \in Q$ should be as simple as possible when Q is both complete and concise.

Clearly "completeness" is mandatory, while the others are just "desirable." Since the possible number of SQL queries on a given database is infinite, we restrict our attention to concise complete subsets. However, there is still infinite number of sets Q that are both complete and concise. Simpler queries are preferred over more complex ones, because it is easier to obtain correct values for cardinalities required

by computing quantities such as n_t and n_o in Example 2.2 (getting exact values for such cardinalities may be difficult for operators embedded in deep query trees).

2.3.2 Implementation

We designed the 5 calibration queries for the PostgreSQL optimizer as follows. We chose queries q_i for Q by introducing individual cost units one by one:

- *cpu_tuple_cost*: We use query q_1:

  ```
  SELECT * FROM R
  ```

  as the calibration query. The relation `R` is first paged into the buffer pool, and hence there is no I/O cost involved: $\mathbf{n}_1 = (0, 0, n_{t1}, 0, 0)^\intercal$.

- *cpu_operator_cost*: We use query q_2:

  ```
  SELECT COUNT(*) FROM R
  ```

  as another calibration query. We then use the method illustrated in Example 2.2. Again, `R` is memory resident: $\mathbf{n}_2 = (0, 0, n_{t2}, 0, n_{o2})^\intercal$ and $n_{t2} = n_{t1}$.

- *cpu_index_tuple_cost*: We use query q_3:

  ```
  SELECT * FROM R WHERE R.A < a
  ```

  where `R.A` has a clustered index and we pick `a` so that the optimizer will choose an index scan. This query involves *cpu_tuple_cost*, *cpu_index_tuple_cost*, and *cpu_operator_cost*. Once again, `R` is memory resident: $\mathbf{n}_3 = (0, 0, n_{t3}, n_{i3}, n_{o3})^\intercal$.

- *seq_page_cost*: We use query q_4:

  ```
  SELECT * FROM R
  ```

  as the calibration query. This query will be executed in a sequential scan, and the cost model only involves overhead in terms of *seq_page_cost* and *cpu_tuple_cost*: $\mathbf{n}_4 = (n_{s4}, 0, n_{t4}, 0, 0)^\intercal$.

- *random_page_cost*: We use query q_5:

```
SELECT * FROM R where R.B < b
```

as the calibration query. Here `R.B` is some attribute of the relation `R` on which an *unclustered* index is built. The values of `B` are uniformly generated, and we pick `b` so that the optimizer chooses an index scan. Ideally, we would like that the qualified tuples were *completely* randomly distributed so that we could isolate the parameter *random_page_cost*. However, in practice, *pure* random access is difficult to achieve, since the execution subsystem can first determine the pages that need to be accessed based on the qualified tuples before it actually accesses the pages. In this sense, local sequential accesses are unavoidable, and the query plan involves more or less overhead in terms of *seq_page_cost*. In fact, a typical query plan of this query will contain all the five parameters: $\mathbf{n}_5 = (n_{s5}, n_{r5}, n_{t5}, n_{i5}, n_{o5})^\intercal$.

Notice that $\mathbf{n}_i$ can be estimated relatively accurately due to simplicity of q_i. Furthermore, the 5 equations generated by the 5 queries are *independent*, which guarantees the existence of a unique solution for $\mathbf{c}$. This can be easily seen by observing the matrix $\mathbf{N} = (\mathbf{n}_1, \cdots, \mathbf{n}_5)^\intercal$, namely,

$$\mathbf{N} = \begin{pmatrix} 0 & 0 & n_{t1} & 0 & 0 \\ 0 & 0 & n_{t2} & 0 & n_{o2} \\ 0 & 0 & n_{t3} & n_{i3} & n_{o3} \\ n_{s4} & 0 & n_{t4} & 0 & 0 \\ n_{s5} & n_{r5} & n_{t5} & n_{i5} & n_{o5} \end{pmatrix}.$$

Note that the determinant $|\mathbf{N}|$ satisfies $|\mathbf{N}| \neq 0$, since by rearranging the columns of $\mathbf{N}$, we can make it a triangular matrix.

To make this approach more robust, our implementation uses multiple queries for each q_i and finds the best-fitting of $\mathbf{c}$. This is done by picking different relations `R` and different values for the `a` in the predicates of the form `R.A < a`.

2.4 Refining Cardinality Estimation

We discuss how to refine **n** in this section. To make this chapter self-contained, we first discuss how the optimizer obtains **n** for a given query plan. We then propose a sampling-based method of refining the cardinality estimates (and hence the **n**) of the final plan chosen by the optimizer. We describe the details of the algorithm and our current implementation.

2.4.1 Optimizer's Estimation of n

The optimizer estimates query execution cost by aggregating the cost estimates of the operators in the query plan. To distinguish blocking and non-blocking operators, this cost model comprises of the *start_cost* and *total_cost* of each operator:

- *start_cost* (sc) is the cost before the operator can produce its first output tuple;
- *total_cost* (tc) is the cost after the operator generates all of its output tuples.

Note that the cost of an operator includes the cost of its child operators.

As an example, we show how **n** is derived for the *in-memory sort* and *nested-loop join* operators in PostgreSQL. These operators are representative of blocking and non-blocking operators, respectively. In the following illustration, *run_cost* (rc for short) is defined as $rc = tc - sc$, and N_t is the (estimated) number of input tuples for the operator. Observe that the costs are given as linear combinations of **c**.

Example 2.3 (In-Memory Sort)**.** Quick sort *is used for tables that optimizer estimates can be completely held in memory. The values* sc *and* rc *are estimated as follows:*

$$\begin{aligned} sc &= 2 \cdot c_o \cdot N_t \cdot \log N_t + tc \text{ of child}, \\ rc &= c_t \cdot N_t. \end{aligned}$$

Example 2.4 (Nested-Loop Join)**.** *The* nested-loop join *operator joins two input relations. The* sc *and* rc *are estimated as follows:*

$$\begin{aligned} \text{sc} &= \text{sc } \textit{of outer child} + \text{sc } \textit{of inner child}, \\ \text{rc} &= c_t \cdot N_t^o \cdot N_t^i + N_t^o \cdot \text{rc } \textit{of inner child}. \end{aligned}$$

N_t^o *and* N_t^i *are the number of input tuples from the outer and inner child operator.*

Notice that the main uncertainty in $\mathbf{n}$ comes from the estimated input cardinality N_t in both of these examples. In general, the logic flow in the cost models of PostgreSQL optimizer can be summarized with five steps:

1) estimate the input/output cardinality;

2) compute the CPU cost based on cardinality estimates;

3) estimate the number of accessed pages according to the cardinality estimates;

4) compute the I/O cost based on estimates of accessed pages;

5) compute the total cost as the sum of CPU and I/O cost.

Hence, our main task in refining $\mathbf{n}$ is to refine the input/output cardinalities for each physical operator in a given query plan.

2.4.2 Cardinality Refinement

As mentioned in the introduction, traditionally cardinality estimation has had to satisfy strict performance constraints because it is done for every plan considered by the optimizer. This has led to compromises that may produce inaccuracies in estimates that are too large for execution time estimation.

Our goal is to *refine* the cardinality estimates for the plan chosen by the optimizer. Clearly, this will increase the overhead of the optimization phase. However, the key insight here is that because the refinement procedure only needs to be performed *once* per query, rather than once per plan, we can afford to spend more time than is possible for traditional cardinality estimation.

2.4.3 A Sampling-Based Approach

In principle, any approach that can improve cardinality estimation can be applied. We use a generalized version of the *sequential-sampling* estimator proposed in [44] for the following two reasons:

- It incorporates a tunable trade-off between efficiency (i.e., the number of samples taken) and effectiveness (i.e., the precision of the estimates);
- It can simultaneously estimate cardinalities for multiple operators in the plan.

In this chapter, we extend the framework in [44] in the following two aspects:

- The estimator described in [44] is for join queries. We generalize the framework to queries with arbitrary number of selections and joins. We prove that this extension preserves the two key properties, namely, *unbiasedness* and *strong consistency*, of the original estimator.
- The framework described in [44] uses random disk accesses to take samples. In comparison, we propose to take samples offline, store them as materialized views, and directly use them at runtime. This greatly reduces the runtime overhead and requires very minimal changes to the database engine (e.g., a few hundred lines of C code in the case of PostgreSQL). We further show that this offline sampling preserves the semantics of the original online sampling.

We next first describe the estimator in its generalized form, and then describe our cardinality refinement algorithm and its implementation details.

2.4.4 The Estimator

Let $\mathcal{D}$ be a database consisting of K relations R_1, ..., R_K. Suppose that R_k is partitioned into m_k blocks each with size N_k, namely, $|R_k| = m_k N_k$. Consider the two basic relational operators: *selection* σ_F (F is a boolean formula representing the selection condition), and *cross-product* $\times$. For σ_F, we define the output of an input block B to be $\sigma_F(B)$. For $\times$, we define the output of the two input blocks B and B' to be $B \times B'$.

Instead of estimating the cardinality of the output relation directly, the estimator will estimate the *selectivity* of the operator, which is defined as the output cardinality divided by the input cardinality. Specifically, the selectivity of the selection operator σ_F is $\rho_R = |\sigma_F(R)|/|R|$ where R is the input relation. Moreover, the selectivity of σ_F on a particular block B of R is $\rho_B = |\sigma_F(B)|/|B|$. On the other hand, the selectivity of the cross-product operator $\times$ is always 1. It is then straightforward to obtain the output cardinality once we know the selectivity of the operator.[1]

In the following, without loss of generality, we will assume that the query considered is over relations R_1, ..., R_K, and we use the notation $\mathbf{R} = R_1 \times \cdots \times R_K$. Let $B(k, j)$ be the j-th block of relation k ($1 \leqslant j \leqslant m_k$, and $1 \leqslant k \leqslant K$). We use $\mathbf{B}(L_{1,i_1}, ..., L_{K,i_K})$ to represent $B(1, L_{1,i_1}) \times \cdots \times B(K, L_{K,i_K})$, where $B(k, L_{k,i_k})$ is the block (with index L_{k,i_k}) randomly picked from the relation R_k in the i_k-th sampling step. Moreover, we use the notation $\mathbf{B}_i$ if $i_1 = i_2 = \cdots = i_K = i$.

We first have the following observation:

Lemma 2.5. *Consider* $\sigma_F(\mathbf{R})$. *Let* $\mathbf{B}_1$, ..., $\mathbf{B}_n$ *be a sequence of* n *random samples (with replacement) from* $\mathbf{R}$. *Define* $\rho_{\mathbf{B}_i} = |\sigma_F(\mathbf{B}_i)|/|\mathbf{B}_i|$ *(*$1 \leqslant i \leqslant n$*). Then* $E\left[\rho_{\mathbf{B}_i}\right] = \rho_{\mathbf{R}}$.

Proof. We have

$$\rho_{\mathbf{R}} = \frac{|\sigma_F(\mathbf{R})|}{|\mathbf{R}|} = \frac{\sum_{k=1}^{K} \sum_{j=1}^{m_k} |\sigma_F(B(k,j))|}{\prod_{k=1}^{K} m_k N_k}.$$

Since

$$E\left[\left|\sigma_F(\mathbf{B}_i)\right|\right] = \frac{\sum_{k=1}^{K} \sum_{j=1}^{m_k} |\sigma_F(B(k,j))|}{\prod_{k=1}^{K} m_k},$$

[1]Note that the input cardinality is already known before the estimation procedure runs. It is simply the product of the cardinalities of the underlying relations that are inputs to the operator, which can be directly obtained from the statistics stored in system catalogs.

it then follows that

$$\mathrm{E}\left[\rho_{\mathbf{B}_i}\right] = \frac{\mathrm{E}\left[\left|\sigma_F(\mathbf{B}_i)\right|\right]}{\prod_{k=1}^{K} N_k} = \frac{\sum_{k=1}^{K}\sum_{j=1}^{m_k}|\sigma_F(B(k,j))|}{\prod_{k=1}^{K} m_k N_k} = \rho_{\mathbf{R}}.$$

This completes the proof of the lemma. □

Define

$$\tilde{\rho}_{\mathbf{R}} = \frac{1}{n}\sum_{i=1}^{n}\rho_{\mathbf{B}_i}.$$

Then it is easy to see from Lemma 2.5 that $\mathrm{E}\left[\tilde{\rho}_{\mathbf{R}}\right] = \rho_{\mathbf{R}}$. Moreover, since the random variables $\rho_{\mathbf{B}_j}$ are i.i.d., by the strong law of large numbers, we have $\Pr\left[\lim_{n\to\infty}\tilde{\rho}_{\mathbf{R}} = \rho_{\mathbf{R}}\right] = 1$. We summarize this result in the following lemma:

Lemma 2.6. $\mathrm{E}\left[\tilde{\rho}_{\mathbf{R}}\right] = \rho_{\mathbf{R}}$, *and* $\Pr\left[\lim_{n\to\infty}\tilde{\rho}_{\mathbf{R}} = \rho_{\mathbf{R}}\right] = 1$.

Lemma 2.6 can be further generalized to queries that contain an arbitrary number of selections and joins:

Theorem 2.7. *Let* q *be any query involving only selections and joins over* $\mathbf{R}$, *and let* ρ_q *be the selectivity of* q. *Then* $\mathrm{E}\left[\tilde{\rho}_q\right] = \rho_q$, *and* $\Pr\left[\lim_{n\to\infty}\tilde{\rho}_q = \rho_q\right] = 1$.

Proof. The proof is easy by noticing that q can be written as its normal form [7]: $\sigma_F(\mathbf{R})$. We then apply Lemma 2.5 to complete the proof. □

In statistical terminology, the estimator $\tilde{\rho}_q$ is *unbiased*, and *strongly consistent* for ρ_q: the more samples we take, the closer $\tilde{\rho}_q$ is to ρ_q. This gives us a way to control the trade-off between the estimation accuracy and the number of samples we take.

The estimator we just described takes samples from each relation uniformly and independently (called *independent sampling* [44]). Therefore, after n steps, we have n observations in total. In [44], the authors further discussed another alternative called *cross-product* sampling. The idea is that, at the i-th step, assuming the K blocks taken from the K relations are $B(1, L_{1,i})$, ..., $B(K, L_{K,i})$, we can actually join each $B(k, L_{k,i})$ with each $B(k', L_{k',i'})$ such that $1 \leqslant i' \leqslant i$ and $k' \neq k$ (note that in

the case of independent sampling, we only join among the blocks with $i' = i$). In this way, we can obtain n^K observations after n steps.

Define

$$\tilde{\rho}_{\mathbf{R}}^{cp} = \frac{1}{n^K} \sum_{i_1=1}^{n} \cdots \sum_{i_K=1}^{n} \rho_{\mathbf{B}(L_{1,i_1},\cdots,L_{K,i_K})}. \tag{2.2}$$

From Lemma 2.5, it is clear that $\tilde{\rho}_{\mathbf{R}}^{cp}$ is still unbiased, i.e., $E\left[\tilde{\rho}_{\mathbf{R}}^{cp}\right] = \rho_{\mathbf{R}}$. However, since now the $\rho_{\mathbf{B}(L_{1,i_1},\cdots,L_{K,i_K})}$'s are no longer independent, we cannot directly apply the strong law of large numbers to show the strong consistency of $\tilde{\rho}_{\mathbf{R}}^{cp}$, although it still holds here (see Appendix A.1 for the proof):

Lemma 2.8. $E\left[\tilde{\rho}_{\mathbf{R}}^{cp}\right] = \rho_{\mathbf{R}}$, *and* $Pr\left[\lim_{n\to\infty} \tilde{\rho}_{\mathbf{R}}^{cp} = \rho_{\mathbf{R}}\right] = 1$.

Therefore, Theorem 2.7 still holds in the case of cross-product sampling. It is also shown that cross-product sampling is always superior to independent sampling because of its lower sample variance (see Theorem 2 of [44]). Therefore, our cardinality refinement algorithm discussed next is based on cross-product sampling instead of independent sampling.

2.4.5 The Cardinality Refinement Algorithm

There are several considerations when designing our refinement algorithm based on the sampling-based estimator.

First, the estimator needs to access disk to take samples. However, since samples should be randomly taken, this means significant random reads may be required during the sampling phase, which may be too costly in practice. To overcome this issue, as has been suggested in previous applications of sampling in DBMS (e.g., [75]), we take samples offline and store them as materialized views in the database. We found in our experiments that the number of samples required is quite small and therefore can be cached in memory during runtime.

Second, the estimator we discussed so far focuses on estimating the selectivity (or equivalently, cardinality) for a single operator. However, in practice, a query plan may contain more than one operator, and for our purpose of refining the

cost estimates of this plan, we need to estimate the cardinality for each operator. Another good property of the estimator is that, for a query plan with a fixed join order, which is always the case when refinement is performed, we can estimate all selection and join operators in the plan simultaneously. Consider, for example, a three-way join query $q = R_1 \bowtie R_2 \bowtie R_3$. We need to estimate the cardinality for both $q' = R_1 \bowtie R_2$ and q. However, after we are done with q', we can estimate for q by directly evaluating $q' \bowtie R_3$. This means, we can estimate the cardinality for each operator by simply invoking the query plan q over the sample relations and then apply the estimator to each operator (see Theorem 2.9 below).

Theorem 2.9. *Let* $q = \sigma_F(R_1 \times \cdots \times R_K)$ *be an arbitrary query with only selections and joins. For every subquery* $q_i = \sigma_{F_i}(R_1 \times \cdots \times R_i)$ *(*$1 \leqslant i \leqslant K$*, and* F_i *is the selection condition only involving* R_1*, ...,* R_i*),* $E\left[\tilde{\rho}_{q_i}\right] = \rho_{q_i}$ *and* $Pr\left[\lim_{n\to\infty} \tilde{\rho}_{q_i} = \rho_{q_i}\right] = 1$.

Proof. The proof is simply applying Theorem 2.7 to each q_i. □

Third, while the estimator discussed above is both unbiased and strongly consistent, it only works for queries involving selections and joins. In practice, SQL queries can contain additional operators. A particularly common class of such operators we encountered in TPC-H queries is *aggregates* (i.e., *group-by*s), for which we need to estimate the number of distinct values in the input relation. Aggregates basically *collapse* the underlying data distribution, so the estimator cannot work for queries containing aggregates. As a result, we can only apply the sampling-based estimator to the part of the query plan that does not involve aggregates. For aggregate operators, we simply rely on PostgreSQL's models for estimating output cardinalities. However, note that, since the refinement phase may change the input estimates for the aggregate, the output estimates for the aggregate may change as well. We observed in our experimental evaluation (see Section 2.5) that the current approach already leads to promising prediction of execution time in practice. We leave the problem of further integrating state-of-the-art estimators (e.g., the GEE estimator in [19]) for estimating the number of distinct values as future work.

Our cardinality refinement algorithm is illustrated in Algorithm 1. For the input query q, we first call the optimizer to obtain its query plan P_q (line 30). We then

Algorithm 1: Cardinality Refinement.

Input: q, a SQL query
Output: P_q, query plan of q with refined cardinalities

HasAgg $\leftarrow$ *False*;

EstimateCardinality(O):
$P_O \leftarrow \texttt{GetSubPlan}(O)$;
$N_s \leftarrow \prod_{R^s \in \texttt{SampleRelations}(P_O)} |R^s|$;
$E_s \leftarrow \texttt{CardinalityBySampling}(O)$;
$N_O \leftarrow \prod_{R_O \in \texttt{Relations}(P_O)} |R_O|$;
$E_O \leftarrow N_O \cdot \frac{E_s}{N_s}$;
Treat E_O as the cardinality estimate for O;

RecomputeCardinality(O):
if O *has left child* O_{lc} **then**
  $\texttt{EstimateCardinality}(O_{lc})$;
end
if O *has right child* O_{rc} **then**
  $\texttt{EstimateCardinality}(O_{rc})$;
end
if *HasAgg* **then**
  Use optimizer's model to estimate for O;
else
  if O *is aggregate* **then**
    Use optimizer's model to estimate for O;
    HasAgg $\leftarrow$ *True*;
  else
    $\texttt{EstimateCardinality}(O)$;
  end
end

Main:
$P_q \leftarrow \texttt{GetPlanFromOptimizer}(q)$;
foreach $R \in \texttt{Relations}(P_q)$ **do**
  Replace R with its sample relation R^s;
end
Run the plan P_q over the sample relations;
$O \leftarrow \texttt{GetRootOperator}(P_q)$;
$\texttt{RecomputeCardinality}(O)$;
return P_q;

modify P_q by replacing the relations it touches with the corresponding sample relations (i.e., materialized views), and run P_q over the sample relations (line 31 to 34). After that, we call the procedure RecomputeCardinality to refine the cardinality estimation for each operator in P_q (line 36).

The procedure RecomputeCardinality (line 11 to 27) works as follows. It first invokes EstimateCardinality on the child operators (if any) of the current operator O (line 12 to 17). It then checks the flag *HasAgg*, which indicates whether O has any *descendant* operator that is an aggregate. If the flag is set, then it simply calls the optimizer's own model to do cardinality estimation for O (line 18 to 19), since our estimator refinement cannot be applied in this case, as discussed above. If, on the other hand, the flag is not set, then it further checks whether O is itself an aggregate. If so, it again calls the optimizer's cardinality estimation model for O, and sets the flag *HasAgg* (line 21 to 23). If not, it invokes EstimateCardinality to estimate the cardinality of O (line 25). Note that due to the order that EstimateCardinality is invoked, the estimator is applied to each operator in a bottom-up manner. This guarantees that the input cardinality of any operator will be estimated after its child operators (if any). While this is not necessary if we only need to refine the cardinalities, it is necessary since we need to further estimate based on the cardinality (for example, the number of pages accessed), which enforces the same bottom-up ordering here since the cost of an operator covers the cost of its child operators as well (recall Example 2.3 and 2.4).

The procedure EstimateCardinality (line 3 to 9) implements the estimator. GetSubPlan returns the subtree P_O of the query plan with the current operator O as the root. N_s is the product of the cardinalities of the sample relations involved in P_O, and E_s is the exact output cardinality of O when the plan is evaluated over sample relations. Therefore, the estimated selectivity is $\frac{E_s}{N_s}$, and the output cardinality of O over the original relations is then $N_O \cdot \frac{E_s}{N_s}$, where N_O is the product of the cardinalities of the original input relations. In Theorem 2.10, we further show that EstimateCardinality is a particular implementation of the estimator conforming to the semantics of cross-product sampling, with a special *tuple-level* partitioning scheme, where each block contains only a single tuple of the relation.

Theorem 2.10. *The procedure* `EstimateCardinality` *estimates the cardinality of the operator* O *according to the semantics of cross-product sampling.*

Proof. Suppose the query $q = \sigma_F(R_1 \times \cdots \times R_K)$. Consider the following *tuple-level* partitioning scheme: for each relation R_k ($1 \leqslant k \leqslant K$), treat each tuple of R_k as a single partition. In this case, $m_k = |R_k|$ and $N_k = 1$. Since

$$\begin{aligned} E_s &= |\sigma_F(R_1^s \times \cdots \times R_K^s)| \\ &= \sum_{i_1=1}^{|R_1^s|} \cdots \sum_{i_K=1}^{|R_K^s|} |\sigma_F(\{t_{1,i_1}\} \times \cdots \times \{t_{K,i_K}\})|, \end{aligned}$$

where t_{k,i_k} is the tuple (i.e., block) from R_k taken in the i_k-th sampling step, and $N_s = \prod_{k=1}^{K} |R_k^s|$, we estimate

$$\tilde{\rho}_q = \frac{E_s}{N_s} = \frac{\sum_{i_1=1}^{|R_1^s|} \cdots \sum_{i_K=1}^{|R_K^s|} |\sigma_F(\{t_{1,i_1}\} \times \cdots \times \{t_{K,i_K}\})|}{\prod_{k=1}^{K} |R_k^s|}.$$

This has the same semantics as the cross-product estimator defined in Equation (2.2), except that here the $|R_k^s|$ may be different for different R_k. It is straightforward to extend Equation (2.2) to the case where different relations have different sampling steps (i.e., different n's), and Lemma 2.8 (and hence Theorem 2.7) still holds. □

2.5 Experimental Evaluation

In this section, we describe our experimental settings and report our results.

2.5.1 Experimental Settings

We implemented Algorithm 1 inside PostgreSQL 9.0.4 by modifying the query optimizer. In addition, we added instrumentation code to the optimizer to collect the input cardinalities for each operator. Our software setup was PostgreSQL on Linux Kernel 2.6.18, and we used both TPC-H 1GB and 10GB databases.

Our experiments were conducted on two different hardware configurations:

- *PC1*: configured with a 1-core 2.27GHz Intel CPU and 2GB memory;
- *PC2*: configured with an 8-core 2.40GHz Intel CPU and 16GB memory.

We randomly drew 10 queries from each of the 21 query templates, and we ran each query 5 times.[2] Our error metric is computed based on the mean execution time of the queries. We cleared both the filesystem and DB buffers between each run of each query.

Since the original TPC-H database generator uses uniform distributions, to test the robustness of different approaches on different data distributions, we also used a skewed TPC-H database generator [5]. This database generator populates a TPC-H database using a Zipf distribution. This distribution has a parameter z that controls the degree of skewness. $z = 0$ generates a uniform distribution, and as z increases, the data becomes more and more skewed. We created skewed databases generated using $z = 1$.

2.5.2 Calibrating Cost Units

We use the approach described in Section 2.3 to generate calibration queries. The calibrated values for the 5 PostgreSQL optimizer parameters on PC1 and PC2 are shown in Table 2.1 and 2.2, respectively. Except for *random_page_cost*, the cost units show very small variance when profiled under different relations.

Calibrating the *random_page_cost* is more difficult. As discussed in Section 2.3.2, achieving purely random reads in a query appears difficult in practice. In addition, the number of random pages accessed as estimated by optimizer is based on statistics about correlations between the order of keys stored in the unclustered index and their actual order in the corresponding data file. Therefore, there is some inherent uncertainty in this estimation. It is interesting future work to see whether

[2]We excluded the template Q15 because it creates a view before the query runs, which is not supported by our current implementation of Algorithm 1.

the *random_page_cost* could be calibrated more accurately with different methods than the one described in this chapter.

Optimizer Parameter	Calibrated $\mu \pm \sigma$ (ms)	Default
seq_page_cost	5.53e-2 $\pm$ 3.09e-3	1.0
random_page_cost	6.50e-2 $\pm$ 2.32e-2	4.0
cpu_tuple_cost	1.67e-4 $\pm$ 5.83e-6	0.01
cpu_index_tuple_cost	3.41e-5 $\pm$ 2.30e-5	0.005
cpu_operator_cost	1.12e-4 $\pm$ 1.30e-6	0.0025

Table 2.1: Actual values of PostgreSQL optimizer parameters on PC1.

Optimizer Parameter	Calibrated $\mu \pm \sigma$ (ms)	Default
seq_page_cost	5.03e-2 $\pm$ 3.82e-3	1.0
random_page_cost	4.89e-1 $\pm$ 7.44e-2	4.0
cpu_tuple_cost	1.41e-4 $\pm$ 1.35e-5	0.01
cpu_index_tuple_cost	3.34e-5 $\pm$ 3.85e-5	0.005
cpu_operator_cost	7.10e-5 $\pm$ 1.52e-5	0.0025

Table 2.2: Actual values of PostgreSQL optimizer parameters on PC2.

Note that the default settings of the parameters fail to accurately reflect the actual relative magnitudes. For example, on PC1, the ratio of calibrated *cpu_tuple_cost* to *seq_page_cost* is about 0.003 instead of 0.01.

Clearly, the overhead of this profiling stage depends on how many calibration queries we use. In our experiments on the TPC-H database, we used the 5 largest relations as the `R` in `SELECT * FROM R` and `SELECT COUNT(*) FROM R`, respectively. For `SELECT * FROM R WHERE R.A < a`, we used the largest relation (*lineitem*), and generated 10 queries where the predicate `R.A < a` had different selectivities in each. Under this setting, the profiling stage usually finishes in less than an hour, which is substantially less than the long training stage of machine-learning based approaches reported in previous work.

Moreover, our profiling stage was conducted on top of the uniform TPC-H database. Note that we do not need to run it again for the skewed TPC-H database. This is because the values of the cost units only depend on the specific hardware

configuration. After the cost units are calibrated, they can be used as long as the hardware configuration does not change. On the other hand, machine-learning based approaches usually need to collect new training data and rebuild the predictive model if the underlying data distribution significantly changes.

2.5.3 Prediction Results

We evaluated the accuracy of prediction in terms of the *mean relative error* (MRE), a metric used in [11]. MRE is defined as

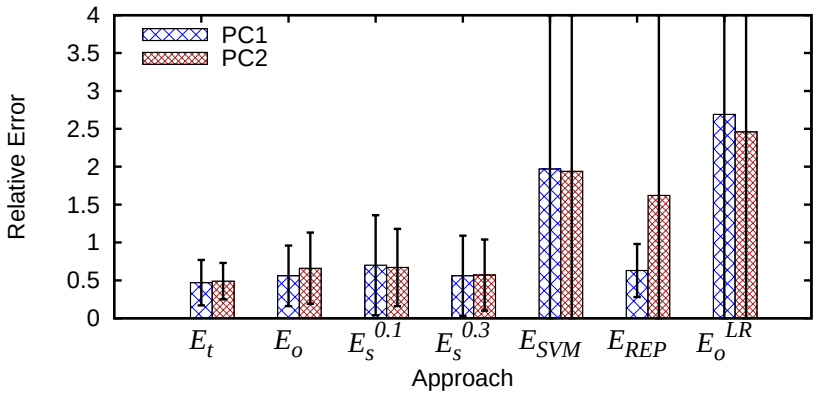
$$\frac{1}{M}\sum_{i=1}^{M}\frac{|T_i^{pred} - T_i^{act}|}{T_i^{act}},$$

where M is the number of testing queries, T_i^{pred} and T_i^{act} are the predicted and actual execution time of the testing query i, respectively.

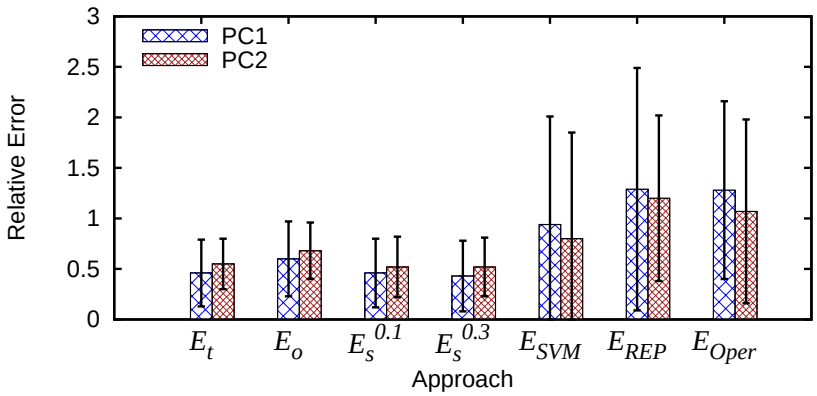
We compare the prediction accuracy of our approach with several state-of-the-art machine-learning based solutions: *plan-level modeling with SVM* [11], *plan-level modeling with REP trees* [96], and *operator-level modeling with Multivariate Linear Regression (MLR)* [11]. We use the same set of features as described in [11]. We focus on the settings of the so-called *dynamic workload* in [11]. The idea of plan-level modeling was also tried in [36], and the authors chose to use Kernel Canonical Correlation Analysis (KCCA) [15] instead of SVM as the machine-learning approach. We do not compare our techniques with theirs, for it has been shown in [11] that both the plan-level and operator-level modeling approach of [11] are superior to the KCCA-based approach for dynamic workloads.

To generate a dynamic workload, we conducted the next "leave-one-template-out" experiment as in [11]. Among the N TPC-H query templates, we chose one template to generate the queries whose execution time is to be predicted, and the other $N-1$ templates were used to generate the training queries used by the machine learning methods to train their predictive models.

Figure 2.2 shows the results on the uniform (i.e., $z = 0$) 1GB TPC-H database. As presented in [11], operator-level modeling requires that the testing queries do not



(a) Without E_{Oper}, 21 templates



(b) With E_{Oper}, 11 templates

Figure 2.2: Uniform TPC-H 1GB database.

contain operators not used by the training queries. Since some TPC-H templates include specific operators not found in the other templates (e.g., *hash-semi-join*), we excluded these templates from our experiments. The same argument applies to TPC-H templates containing PostgreSQL-specific structures (i.e., `INITPLAN` and `SUBQUERY`). The authors of [11] also excluded these queries for the same reason. However, we note here that this is a problem due to the particular choice of the workload and database system, not due to the operator-level modeling itself. If TPC-H were a more varied workload, we would not have this restriction. For example, if it had multiple queries that used the hash-semi-join operator, we could have

incorporated queries with that operator in our experiments. This leaves 11 TPC-H templates participating in the dynamic workload experiment when operator-level modeling is leveraged (see Figure 2.2(b)).

In Figure 2.2, the x-axis represents the approaches we tested in the experiments, and the y-axis shows the average error and the standard deviation (shown with the error bars) over the TPC-H templates. Here, E_t is the prediction error of our approach when the *true* cardinalities are used (the true cardinalities are measured in an artificial "pre-running" of the query — we present this number to provide insight into what could be achievable if we were able to get perfect cost estimates). E_o is the prediction error of our approach when the cardinalities from the optimizer are used without refinement. E_s^f is the prediction error of our approach when the cardinalities are estimated via sampling, where f is the sampling ratio (e.g., $f = 0.1$ means we take a 10% sample from each underlying table). In our experiments, we tested sampling ratios $f = 0.05, 0.1, 0.2, 0.3, 0.4$. Due to space limitations in the plots, we only present the results of $f = 0.1$ and $f = 0.3$. E_{SVM}, E_{REP}, and E_{Oper} are the prediction errors of the three machine-learning based approaches, i.e., plan-level modeling with SVM, plan-level modeling with REP, and operator-level modeling, respectively. Finally, as a baseline, E_o^{LR} is the prediction error of mapping the original cost estimates from the optimizer to the execution time via linear regression, as was done in previous work.

We have several observations. First, in the case of uniform data, the cost models with properly tuned c's already work well (the E_o in Figure 2.2 is close to E_t). Sampling does not help much in improving the prediction accuracy. This is reasonable, because the assumptions like uniformity and independence leveraged by the optimizer for cardinality estimation usually hold in this case.

Second, the performance of machine-learning based approaches is not consistent. For some queries, their predictions are good. However, for the other queries, their predictions are far away from the true values. This can be observed by noticing the big error bars in the figures. As an example, the E_{SVM} in Figure 2.2(a) varies between 0.03 for Q7 and 12.16 for Q17. One possible reason for this is: most machine learning methods assume that the testing queries should be *similar* to the queries

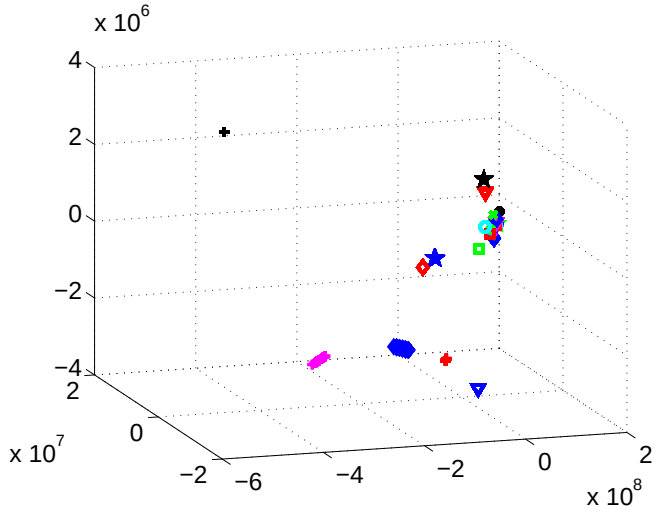


Figure 2.3: Queries projected on the 3 dominating principal components.

used in training the model. More specifically, the feature vectors of the testing queries should be close to those feature vectors of the training queries, in terms of the distance in the feature space. Unfortunately, this assumption is not valid for dynamic workloads.

To see this, we further apply Principal Component Analysis (PCA) [47] on the query features (47 features in total) and project the queries onto the subspace spanned by the three most dominating principal components. It is well known that these dominating components reveal the *major* directions in the feature space. While it is true that more principal components are better, we restrict ourselves to 3-dimensional space for the purpose of visualization.

Figure 2.3 shows the queries in the projected space, where each combination of color and shape represents one query template. From the figure we can see that the templates can be grouped into several clusters. About half of the templates fall into the rightmost cluster, and each of the remaining templates usually forms a singleton cluster. The distances between clusters are quite big. Note that PCA will not increase the distances between feature vectors after the projection, which means the distances between feature vectors in the original 47-dimensional space can only be the same or even bigger. This suggests that there is little similarity among the

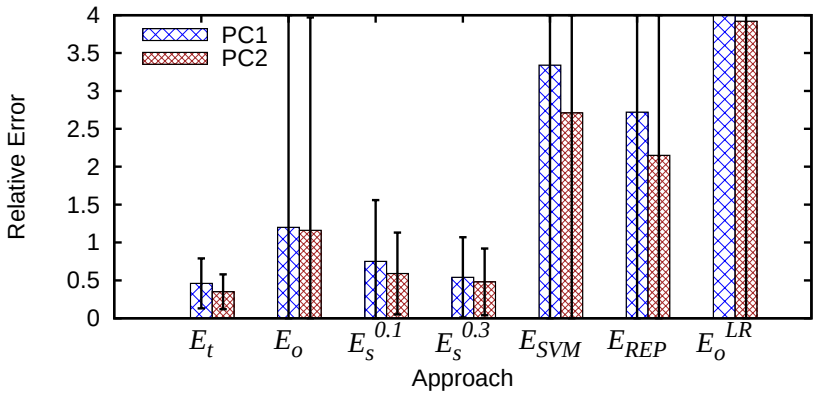
TPC-H templates within different clusters. Therefore, if we test the queries from a template within a singleton cluster, by using the model trained with the other templates, then there is little hope for us to observe good predictions.

Third, machine-learning approaches are sensitive to the set of queries used in training. The prediction errors for some queries fluctuate dramatically when different sets of training queries are used[3]. For instance, E_{REP} for Q8 is 0.44 when 20 templates are used in training (as in Figure 2.2(a)), but it increases to 2.87 when only 10 templates are used (as in Figure 2.2(b)). Picking a set of proper training queries hence is critical in practice when using machine-learning approaches. However, it seems difficult in the environment when workload is not known in advance.

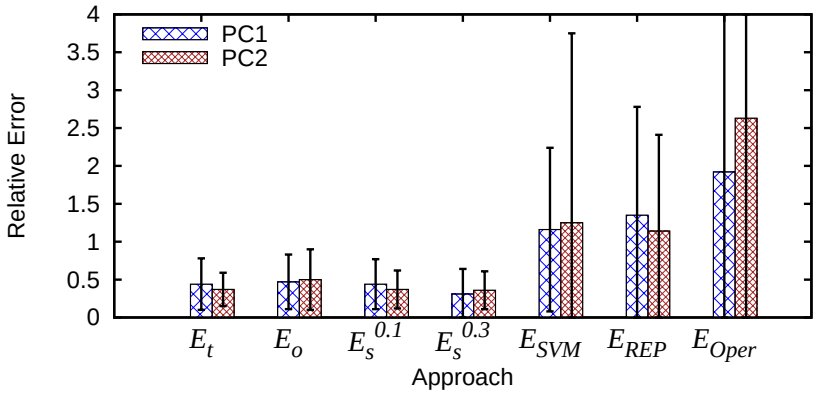
Figure 2.4 further presents the results on the skewed TPC-H 1GB database. As expected, when data becomes skewed, the cardinality estimates from the optimizer become inaccurate, and hence the predictive power is weakened. However, by leveraging the sampling-based cardinality correction, the prediction accuracy is improved. Moreover, more samples usually mean better prediction accuracy, as long as the overhead on sampling is acceptable (see Section 2.5.4). We note that the sampling overhead can be up to 20% on this data set. As we will see, this can be viewed as a problem that arises on small data sets, as the overhead due to sampling for the 10GB data set is much lower. On the other hand, the performance of machine-learning based approaches becomes even worse. This is perhaps partially due to the worse distortion of the assumption of similar training and testing queries.

Similar results on the TPC-H 10GB database are observed in our experiments, as shown in Figure 2.5 and Figure 2.6. Here, to make the overall experiment time controllable, as done in [11], we kill the query if it runs longer than an hour. This leaves us with 18 templates participating in the evaluation. We tested sampling ratios $f = 0.01, 0.02, 0.05, 0.1$, and present the results of $f = 0.02$ and $f = 0.05$, for space constraints. Note that, while the database size scales up by a factor of 10, the required absolute number of samples to achieve predictions close to the ideal case (compare $E_s^{0.05}$ and E_t in the figures) remains almost the same. We need

[3]Recall that, to be fair, we only use 10 templates in training when comparing with operator-level modeling (as in Figure 2.2(b)).



(a) Without E_{Oper}, 21 templates



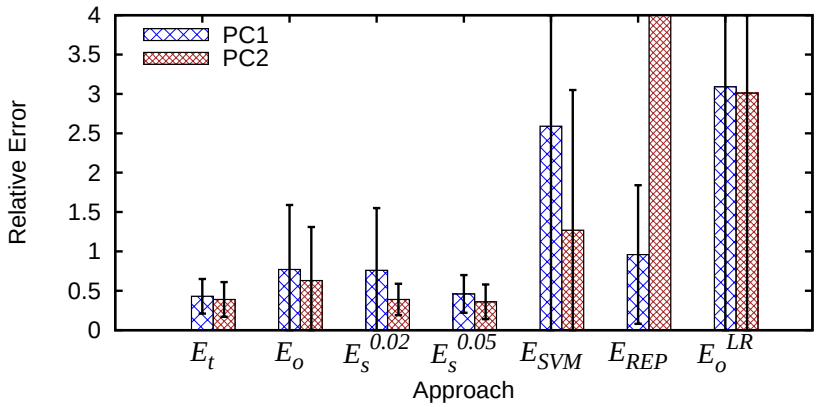
(b) With E_{Oper}, 11 templates

Figure 2.4: Skewed TPC-H 1GB database.

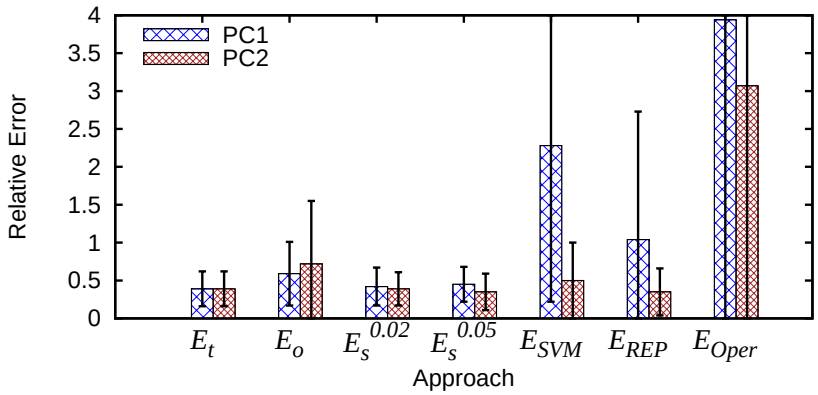
$0.05 \times 10\text{GB} = 0.5\text{GB}$ samples here, while we need $0.3 \times 1\text{GB} = 0.3\text{GB}$ samples in the case of 1GB database. Therefore, the additional overhead of taking samples becomes ignorable when the database is larger (see Section 2.5.4).

2.5.4 Overhead of Sampling

Figure 2.7 shows the additional runtime overhead due to sampling for the 1GB TPC-H database. Here r_s^f is defined as $r_s^f = T_s/T$, where T_s and T are the time to run the queries over the sample tables and original tables, respectively, and f is the



(a) Without E_{Oper}, 21 templates

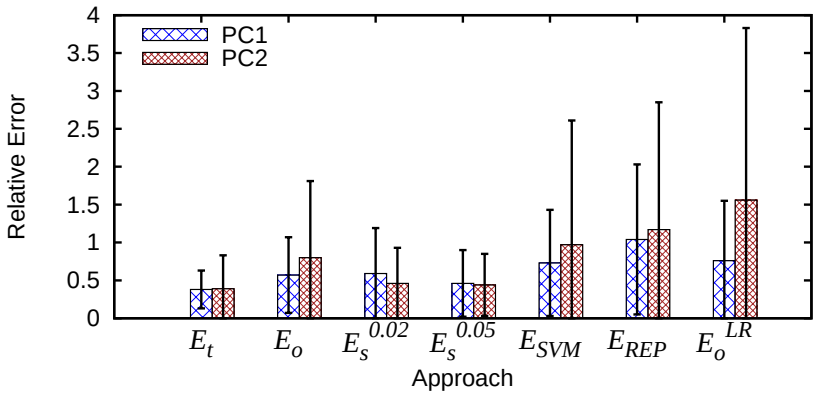


(b) With E_{Oper}, 11 templates

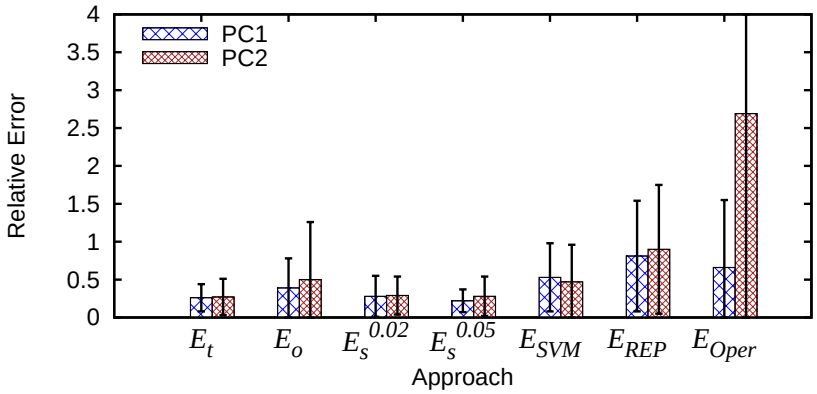
Figure 2.5: Uniform TPC-H 10GB database.

sampling ratio as before. For each sampling ratio, we report the average r_s^f as well as the standard deviation (shown with the error bars) over the TPC-H templates.

We can see that for the sampling ratio $f = 0.3$, which allows us to achieve close prediction accuracy to what if the true cardinalities were used on both the uniform and skewed data, the average additional runtime overhead is around 20% of the actual execution time of the query. Note that for query optimization, 20% is prohibitively high. For example, it means that we can only consider $1/20\% = 5$ plans during optimization before the estimation cost dominates the query execution time, since sampling should be invoked for every query plan considered. But for



(a) Without E_{Oper}, 21 templates

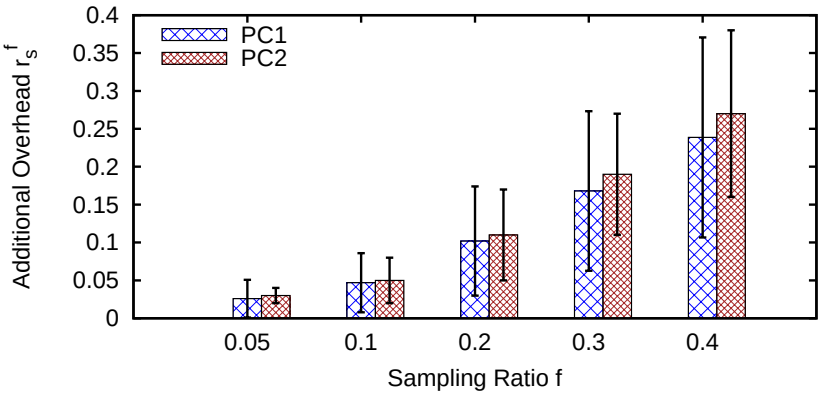


(b) With E_{Oper}, 11 templates

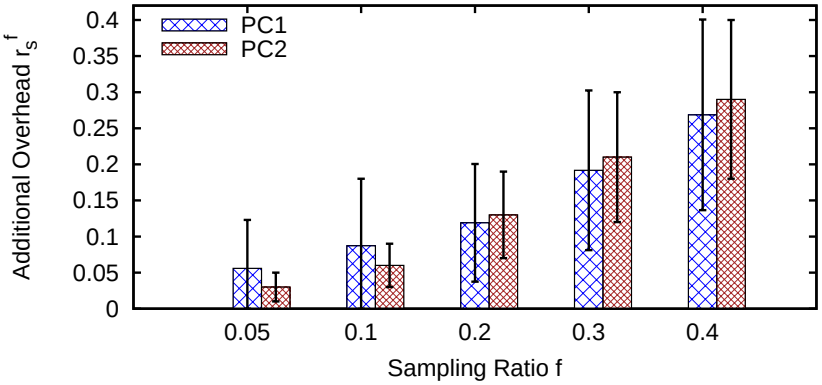
Figure 2.6: Skewed TPC-H 10GB database.

our purposes, where we are trying to estimate the running time of a single query plan, this amount of overhead may be acceptable. Perhaps more importantly, this overhead drops dramatically when we move to the 10GB data set.

Figure 2.8 further presents the results over 10GB TPC-H database. It confirms that the additional overhead introduced by sampling is even smaller, compared with the overhead of running the original query. For the case where good prediction can be achieved (i.e., $f = 0.05$, see Figure 2.5 and Figure 2.6), the additional overhead is below 4% on average. This demonstrates the practicality of incorporating sampling for the purpose of query time prediction.



(a) On uniform data

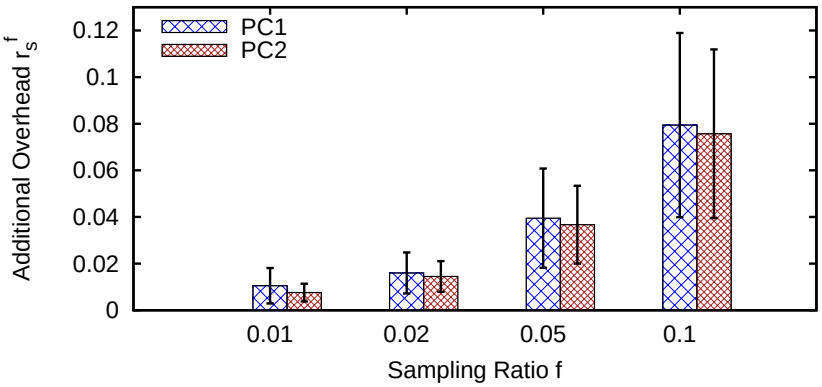


(b) On skewed data

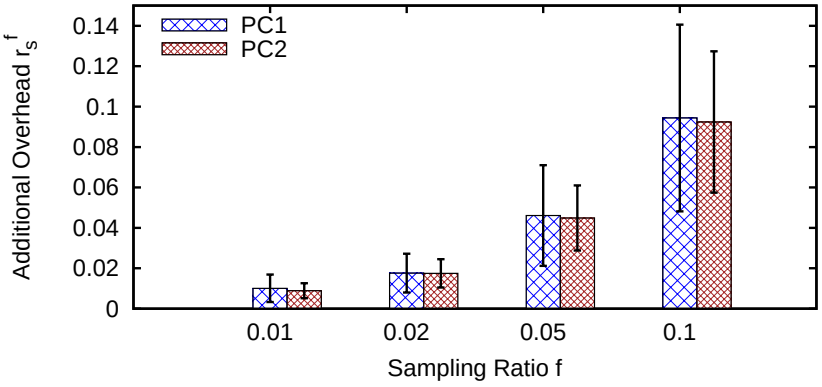
Figure 2.7: Additional overhead of sampling on TPC-H 1GB database.

2.5.5 Discussion

The cost model used by PostgreSQL's optimizer exhibits a nice linearity property, which eases the selection of calibration queries. It remains interesting to study cost models of other database systems and calibration approaches when linearity does not hold. On the other hand, PostgreSQL's cost model is by no means perfect. For example, it does not consider the differences between the CPU overheads of processing different attributes in a tuple, which might be significantly different. We expect that a better cost model can further improve prediction accuracy but it is beyond the scope of this chapter.



(a) On uniform data



(b) On skewed data

Figure 2.8: Additional overhead of sampling on TPC-H 10GB database.

As another remark, the amount of samples required to achieve certain degree of estimation accuracy depends on the magnitude of the selectivity estimate as well as its variance [44]: in general, smaller selectivity and larger variance require larger sample size. Note that the sample size here is *absolute* rather than relative, and it does not depend on the database size. Therefore, a larger database does not mean more samples are necessary, as we have seen in the case of TPC-H. Nonetheless, it is a challenging problem to determine a good sample size for a given database and workload a priori, and we leave it as one direction for future exploration.

2.6 Related Work

Query optimizers have built-in cost models that provide cardinality/cost estimates for a given query. There is a lot of previous work on this topic, including methods based on sampling (e.g., [49, 62]), methods based on histograms (e.g., [53]), methods based on machine learning (e.g., [37, 89]), and methods based on using execution feedback (e.g., [24, 85]). However, the purpose of these estimates is to help the optimizer pick a relatively good plan, not to predict the *actual* execution time of the query. Therefore, these estimates need not to be very accurate as long as the optimizer can leverage them to distinguish good plans from bad ones. As shown in [11, 36], without proper calibration, directly leveraging these cost estimates cannot provide good predictions of execution time. Nonetheless, it is interesting future work to see the effectiveness by incorporating some methods other than sampling into our current framework for refining cardinality estimates. For example, recent work [89] presented an efficient approach based on graphical models, which was reported to have an order of magnitude better selectivity estimates.

Another related research direction is *query progress indicators* [23, 57, 61, 64]. The task of a progress indicator is to dynamically monitor the percentage of work that has been done so far for the query. The key difference from query execution time prediction is that progress indicators usually rely on *runtime* statistics obtained *during* the actual execution of the query, which are not available if the prediction is restricted to be made before query execution. A query running time predictor could be useful in providing the very first estimate for a progress indicator (one used before the query starts executing).

Quite surprisingly, the problem of predicting *actual* execution time of a query has been specifically addressed only recently [36]. Existing work [9, 11, 33, 36, 88] usually employs predictive frameworks based on statistical machine learning techniques. The focus of [9, 33, 88] is query execution time prediction for multi-query workloads, and we will discuss our work on this problem in the next chapter.

In [36], each query is represented as a set of features containing an instance count and cardinality sum for each possible operator. Kernel Canonical Correlation

Analysis (KCCA) [15] modeling techniques are then used to map the queries from their feature space onto their performance space. One main limitation of this approach is that its prediction is based on taking the average of the k (usually 3) nearest neighbors in the training set, which means that the prediction can never exceed the longest execution time observed during training stage. Hence, when the query to be predicted takes significantly longer time than all the training queries observed, the model is incapable of giving reasonable predictions.

In [11], a similar idea of using features extracted from the entire plan to represent a query is leveraged, and the authors propose to use SVM instead of KCCA. However, the SVM approach still suffers from the same generalization problem. To alleviate this, the authors further apply this idea at the operator-level. But from the reported experimental results, it seems that operator-level modeling is still quite vulnerable to workload changes. Our approach in this chapter avoids this generalization problem, for it does not rely on any particular training queries.

2.7 Summary

In this chapter, we studied the problem of leveraging optimizer's cost models to predict query execution time. We show that, after proper calibration, the current cost models used by query optimizers can be more effective for predicting query execution time than reported by previous work.

Of course, it is possible that a new machine learning technique, perhaps with improved feature selection, will outperform the techniques presented here. On the other hand, further improvements are also possible in optimizer-based running time prediction. Perhaps the most interesting aspect of this work is the basic question it raises: should query running time prediction treat the DBMS as a black box (the machine learning approach), or should we exploit the fact that we actually know exactly what is going on inside the box (the optimizer based approach)? We regard this chapter as an argument that the latter approach shows promise, and expect that exploring the capabilities of the two very different approaches will be fertile ground for future research.

Chapter 3

Query Execution Time Prediction for Multi-Query Workloads

Predicting execution time for concurrent queries is arguably more important than prediction for standalone queries, because database systems usually allow multiple queries to execute concurrently. The existing work on concurrent query prediction [10, 33], however, assumes that the workload is *static*, namely, the queries participating in the workload are known beforehand. While some workloads certainly conform to this assumption (e.g., the *report-generation* workloads described in [9]), others do not. Real-world database workloads can be *dynamic*, in that the set of queries that will be submitted to the system cannot be known a priori.

In this chapter, we consider the more general problem of *dynamic* concurrent workloads. Unlike most previous work on query execution time prediction, our proposed framework is based on analytic modeling rather than machine learning. Extending our idea in the previous chapter, we first use the optimizer's cost model to estimate the I/O and CPU requirements for each pipeline of each query in isolation, and then use a combination queueing model and buffer pool model that merges the I/O and CPU requests from concurrent queries to predict running times. We compare the proposed approach with a machine-learning based approach that is a variant of previous work. Our experiments show that our analytic-model based

approach can lead to competitive and often better prediction accuracy than its machine-learning based counterpart.

3.1 Introduction

Recall the cost model used by PostgreSQL's query optimizer as presented in Example 2.1 of Chapter 2. For multiple concurrently-running queries, one could try to build a generalization of the optimizer's cost model that explicitly takes into account the execution of other queries. For example, it could make guesses as to what might be in the buffer pool; or what fraction of the CPU this query will get at each point in execution; or which sequential I/O's will be converted to random I/O's due to interference, and so forth. But this seems very difficult if not impossible. First, the equations capturing such a complex system will be messy. Second, it requires very detailed knowledge about the exact mix of queries that will be run and how the queries will overlap (in particular, which parts of each query execution will overlap with which parts of other query's execution). This detailed knowledge is not even available in a dynamic system.

Therefore, instead of building sophisticated extensions to the optimizer's cost model, we retain the single query optimizer estimate, but stop at the point where it estimates the counts of the operations required (rather than going all the way to time). We then use a combination queueing model/buffer pool model to estimate how long each concurrently running query will take.

More specifically, we model the underlying database system with a queueing network, which treats hardware components such as disks and CPU cores as *service centers*, and views the queries as *customers* that visit these service centers. The n's of a query are then the numbers of visits it pays to the service centers, and the c's are the times spent on serving one single visit. In queueing theory terminology, the c's are the *residence times per visit* of a customer, and can be computed with the well-known mean value analysis technique [78, 86]. However, the queueing network cannot account for the cache effect of the buffer pool, which might be important

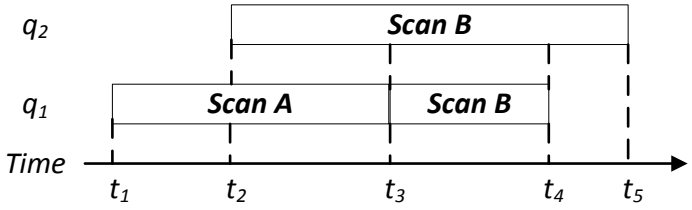


Figure 3.1: Interactions between queries.

for concurrent queries. Therefore, we further incorporate a model to predict buffer pool hit rate [70], based on the "clock" algorithm used by PostgreSQL.

However, queries are not uniform throughout, they change behavior as they go through different phases of their execution. Consider a query q that is concurrently running with other queries. For different operators of q, the cost units (i.e., the c's) might have different values, depending on the particular operators running inside q's neighbors. Consider the following example:

Example 3.1 (Query Interactions)**.** *Figure 3.1 shows two queries* q_1 *and* q_2 *that are concurrently running.* q_1 *starts at time* t_1 *and has 2 scan operators, with the first one scanning the table* A *and the second one scanning the table* B. q_2 *starts at time* t_2 *and has only 1 scan operator that scans the table* B. *The I/O cost units (i.e.,* c_s *and* c_r*) of* q_1 *between* t_2 *and* t_3 *are expected to be greater than that between* t_1 *and* t_2*, due to the contention with* q_2 *on disk service after* q_2 *joins. At time* t_3*, the first scan of* q_1 *finishes, and the second one starts. The I/O cost units of* q_1 *(and* q_2*) are then expected to decrease, since the contention on disk would be less intensive for two scans over the same table* B *than when one is over* A *while the other is over* B*, due to potential buffer pool sharing. At time* t_4*,* q_1 *finishes and* q_2 *becomes the only query running in the system. The I/O cost units of* q_2 *are thus again expected to decrease.*

Therefore, to the queuing/buffer model, a query looks like a series of phases, each with different CPU and I/O demands. Hence, rather than applying the models at the query level, we choose to apply them to each execution phase. The remaining issue is then how to define the "phase" here. One natural choice could be to define a phase to be an individual operator. However, a number of practical difficulties

arise. A serious problem is that database queries are often implemented using an *iterator* model [38]. When evaluating a query, the operators are usually grouped into pipelines, and the execution of operators inside a pipeline are *interleaved* rather than sequential. For this reason, we instead define a phase to be a pipeline. This fixes the issue of "interleaved phases" if a phase were defined as an operator. By doing this, however, we implicitly assumed the requests of a query are relatively constant during a pipeline and may only change at pipeline boundaries. In other words, we use the same c's for different operators of a pipeline. Of course, this sacrifices some accuracy compared with modeling at the operator level, and hence is a tradeoff between complexity and accuracy. Nonetheless, modeling interactions at the pipeline level is still a good compromise between doing it at the operator level and doing it at the query level.

Nonetheless, this still leaves the problem of predicting the future. Throughout the above discussion, we have implicitly assumed that no knowledge is available about queries that will arrive in the future, and our task is to estimate the running times of all concurrently running queries at any point in time. If a new query arrives, the estimates for all other running queries will change to accommodate that query. Of course, if information about future workloads were available we could use that, but this is out of the scope of this chapter.

We have compared our analytic-model based approach with a machine-learning based approach that is a variant of the approach used in [9, 10]. Our experimental evaluation over the TPC-H benchmark shows that, the analytic-model based approach can lead to comparable and often better prediction accuracy than the machine-learning based approach.

The rest of this chapter is organized as follows. We first present our predictive framework and give some analysis in Section 3.2. We then describe the two approaches that combine cost estimates for concurrently-running pipelines in Section 3.3, where Section 3.3.1 describes the machine-learning based approach, and Section 3.3.2 describes the analytic-model based approach, respectively. We present experimental results in Section 3.4, discuss related work in Section 3.5, and summarize this chapter in Section 3.6.

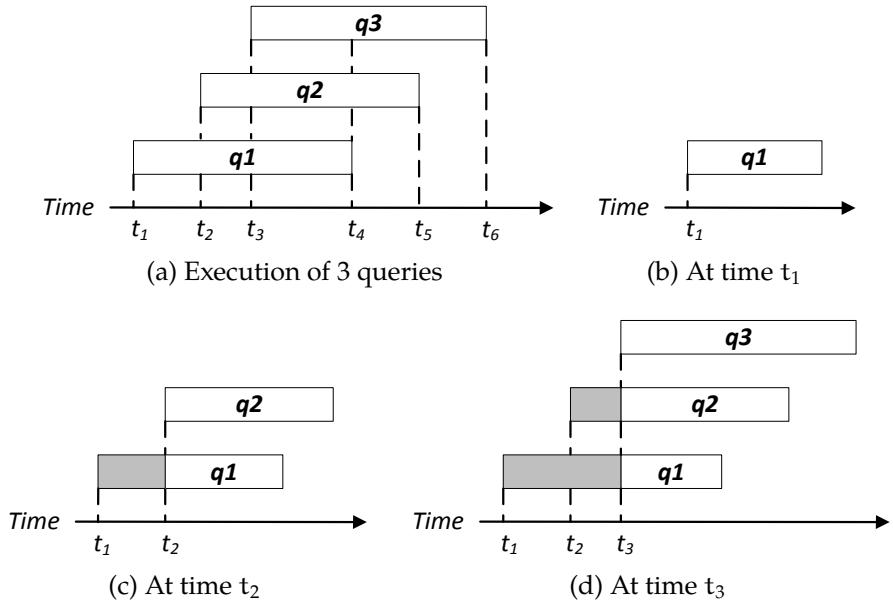


Figure 3.2: The prediction problem for multiple concurrently-running queries.

3.2 The Framework

We present the details of our predictive framework in this section. We first formally define the prediction problem we are concerned with in this chapter, then describe our solution and provide some analysis.

3.2.1 Problem Definition

We use an example to illustrate the prediction problem. As shown in Figure 3.2(a), suppose that we have three queries q_1, q_2, and q_3 that are concurrently running, which arrive at time t_1, t_2, and t_3, respectively. Accordingly, we have three prediction problems in total. At t_1, we need to predict the execution time for q_1 (Figure 3.2(b)). Perfect prediction here would require the information of the upcoming q_2 and q_3, which is unfortunately not available at t_1. So the best prediction for q_1 at t_1 has to be based on assuming that there will be no query coming before q_1 finishes. At t_2, q_2 joins and we need to make a prediction for both q_1 and q_2 (Figure 3.2(c)). For q_1, we actually predict its *remaining* execution time, since it has been running for some

time (the gray part). Perfect predictions would again require the knowledge that q_3 will arrive, which is unavailable at t_2. As a result, the best prediction at t_2 needs the assumption that no query will come before q_1 and q_2 end. The same argument can be further applied to the prediction for q_1, q_2, and q_3 at t_3 (Figure 3.2(d)). We therefore define our prediction problem as:

Definition 3.2 (Problem Definition)**.** *Let* Q *be a mix of* n *queries* $Q = \{q_1, ..., q_n\}$ *that are concurrently running, and assume that no new query will come before* Q *finishes. Let* s_0 *be the start time of these* n *queries, and let* f_i *be the finish time for the query* q_i*. Define* $T_i = f_i - s_0$ *to be the* execution time *of* q_i*. The problem we are concerned with in this chapter is to build a predictive model* $\mathcal{M}$ *for* $\{T_i\}_{i=1}^n$*.*

For instance, the prediction problem in Figure 3.2(d) is generated by setting $Q = \{q_1, q_2, q_3\}$ and $s_0 = t_3$ in the above definition.

3.2.2 Query Decomposition

To execute a given SQL query, the query optimizer will choose an *execution plan* for it. A plan is a tree such that each node of the tree is a physical operator, such as *sequential scan*, *sort*, or *hash join*. Figure 3.3 presents an example query and the execution plan returned by the PostgreSQL query optimizer.

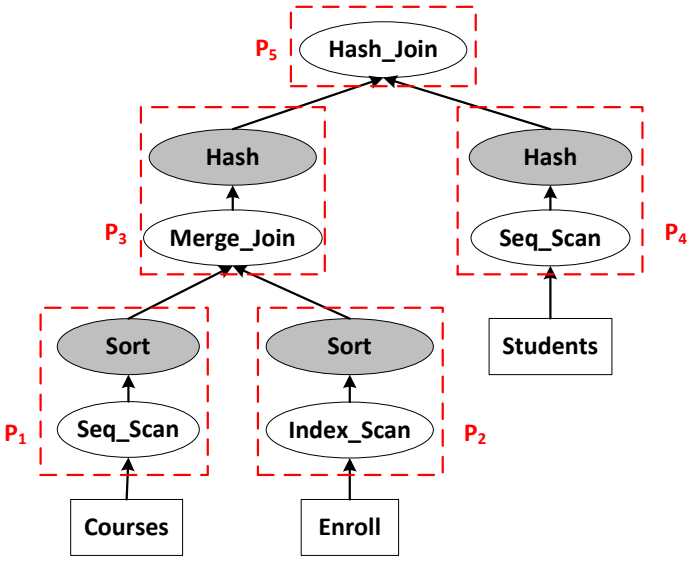
As we have mentioned in Chapter 2, a physical operator can be either *blocking* or *nonblocking*. An operator is blocking if it cannot produce any output tuple without reading all of its input. For instance, the operator *sort* is a blocking operator. In Figure 3.3, blocking operators are highlighted.

Based on the notion of blocking/nonblocking operators, the execution of the query can then be divided into multiple *pipeline*s. As in previous work [23, 64], we define pipelines inductively, starting from the leaf operators of the plan. Whenever we encounter a blocking operator, the current pipeline ends, and a new pipeline starts if any operators are remaining after we remove the current pipeline from the plan. Therefore, a pipeline always ends with a blocking operator (or the root operator). Figure 3.3 also shows the 5 pipelines P_1 to P_5 of the example plan.

Tables:
Students (sid, sname)
Enroll (sid, cid)
Courses (cid, cname)

SELECT S.sid, S.sname
FROM Students S, Enroll E, Courses C
WHERE S.sid = E.sid **AND** E.cid = C.cid
AND S.sid > 1 **AND** S.sid < 10
AND C.cid < 5 **AND** S.sname <> 'Mike'

(a) Database and query



(b) Execution plan

Figure 3.3: Example query and its execution plan.

By organizing concurrently running operators into pipelines, the original plan can also be viewed as a tree of pipelines, as illustrated in Figure 3.3. We assume that at any time, only one pipeline of the plan is running in the database system, which is a common way in current database implementations. The execution plan thus defines a partial order over the pipelines. For instance, in the example plan, the execution of P_1 and P_2 must precede P_3, while the order between P_1 and P_2 is arbitrary. Similarly, the execution of P_3 and P_4 must precede P_5. The execution order of the pipelines can usually be obtained by analyzing the information contained in the plan. For example, in our implementation with PostgreSQL, we order the pipelines based on estimating their start times by using the optimizer's running time estimates. We then decompose the plan into a sequence of pipelines, with

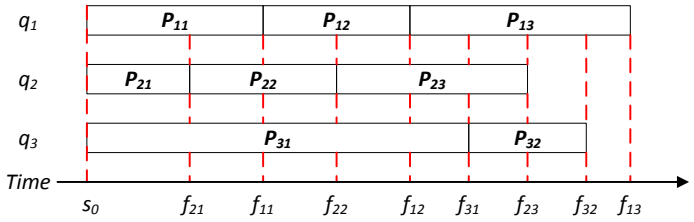


Figure 3.4: Progressive predictor.

respect to their execution order. For the example plan, suppose that the optimizer specifies that P_1 precedes P_2 and P_3 precedes P_4. Then the plan can be decomposed into the sequence of pipelines: $P_1P_2P_3P_4P_5$.

3.2.3 Progressive Predictor

For the given mix of queries q_1, ..., q_n, after decomposing their execution plans into sequences of pipelines, the mix of queries can be viewed as multiple stages of mixes of pipelines. We illustrate this with the following example:

Example 3.3 (Mix of pipelines)**.** *As presented in Figure 3.4, suppose that we have a mix of 3 queries* q_1, q_2, *and* q_3. *After decomposition of their plans,* q_1 *is represented as a sequence of 3 pipelines* $P_{11}P_{12}P_{13}$, q_2 *is represented as a sequence of 3 pipelines* $P_{21}P_{22}P_{23}$, *and* q_3 *is represented as a sequence of 2 pipelines* $P_{31}P_{32}$. *We use* P_{ij} *to denote the* j*th pipeline of the* i*th query, and use* f_{ij} *to denote the time when* P_{ij} *finishes. It is easy to see that whenever a pipeline finishes, we will have a new mix of pipelines. For the example query mix in Figure 3.4, we will thus have 8 mixes of pipelines in total, delimited by the red dash lines that indicate the finish timestamps for the pipelines.*

If we could know the f_{ij}'s, then it would be straightforward to compute the execution time of the P_{ij}'s and hence the q_i. Suppose that we have some model $\mathcal{M}_{ppl}$ to predict the execution time of a pipeline by assuming that its neighbor pipelines do not change. We can then progressively determine the next finishing pipeline and therefore its finish time. For example, in Figure 3.4, we first call $\mathcal{M}_{ppl}$ for the mix of pipelines $\{P_{11}, P_{21}, P_{31}\}$. Based on the prediction from $\mathcal{M}_{ppl}$, we can learn that P_{21} is the next finishing pipeline and we have a new mix of pipelines

$\{P_{11}, P_{22}, P_{31}\}$ at time f_{21}. We then call $\mathcal{M}_{ppl}$ for this new mix again. Note that here we also need to adjust the prediction for P_{11} and P_{31}, since they have been running for some time. We then learn that P_{11} is the next finishing pipeline for this mix and it finishes at time f_{11}. We proceed in this way until all the pipelines finish. The details of this idea are presented in Algorithm 2.

Each pipeline P_{ij} is associated with two timestamps: s_{ij}, the (predicted) start timestamp of P_{ij}; and f_{ij}, the (predicted) finish timestamp of P_{ij}. The (predicted) execution time of P_{ij} is thus $T_{ij} = f_{ij} - s_{ij}$. We also maintain the *remaining ratio* ρ^r_{ij} for P_{ij}, which is the percentage of P_{ij} that has not been executed yet. Algorithm 2 works as follows. For each query q_i, we first call the query optimizer to generate its execution plan $Plan_i$, and then decompose $Plan_i$ into a sequence of pipelines $\mathcal{P}_i$, as illustrated in Section 3.2.2 (lines 1 to 3). The first pipeline P_{i1} in each $\mathcal{P}_i$ is added into the current mix $CurrentMix$. Its start timestamp s_{i1} is set to be 0, and its remaining ratio ρ^r_{i1} is set to be 1.0 (lines 5 to 8).

Algorithm 2 then proceeds stage by stage. It makes a prediction of the initial mix of pipelines by calling the given model $\mathcal{M}_{ppl}$ (line 11). As long as the current mix is not empty, it will determine the pipeline P_{ij} with the shortest (predicted) execution time t_{min}. The current (virtual) timestamp $CurrentTS$ is forwarded by adding t_{min}. The finish time f_{ij} of P_{ij} is then set accordingly, and P_{ij} is removed from the current mix (lines 13 to 16). For each remaining pipeline P_{ik} in the current mix, we update its remaining ratio ρ^r_{ik} by multiplying it by $\frac{t^r_{ik}}{t_{ik}}$, where t_{ik} is the predicted time of P_{ik} (at the beginning time of the current mix of pipelines), and t^r_{ik} is the remaining (predicted) time of P_{ik} when P_{ij} finishes and exits the current mix. $t^r_{ik} = t_{ik} - t_{min}$ by definition (lines 17 to 20). Intuitively, $\frac{t^r_{ik}}{t_{ik}}$ is the *relative* remaining ratio of P_{ik} at the end of the current mix. If $\mathcal{P}_i$ contains more pipelines after P_{ij} finishes, we add the next one $P_{i(j+1)}$ into the current mix, set $s_{i(j+1)}$ to be the current timestamp, and set $\rho^r_{i(j+1)}$ to be 1.0 since the pipeline is just about to start (lines 21 to 23). Note that now the current mix changes, due to removing P_{ij} and perhaps adding in $P_{i(j+1)}$. We thus call $\mathcal{M}_{ppl}$ again for this new mix (line 24). However, we need to adjust the prediction t_{ik} for each pipeline, by multiplying it with its

Algorithm 2: Progressive Predictor.

Input: $Q = \{q_1, ..., q_n\}$, a mix of n SQL queries; $\mathcal{M}_{ppl}$: a model to predict the execution times for a mix of pipelines

Output: $\{T_i\}_{i=1}^{n}$, where T_i is the predicted execution time of the query q_i

for $1 \leqslant i \leqslant n$ **do**
  $Plan_i \leftarrow GetPlan(q_i)$; $\mathcal{P}_i \leftarrow DecomposePlan(Plan_i)$;
end

$CurrentMix \leftarrow \emptyset$;
for $1 \leqslant i \leqslant n$ **do**
  Add $P_{i1} \in \mathcal{P}_i$ into $CurrentMix$; $s_{i1} \leftarrow 0$; $\rho^r_{i1} \leftarrow 1.0$;
end

$CurrentTS \leftarrow 0$;
$MakePrediction(CurrentMix, \mathcal{M}_{ppl})$;
while $CurrentMix \neq \emptyset$ **do**
  $t_{min} \leftarrow MinPredictedTime(CurrentMix)$;
  $CurrentTS \leftarrow CurrentTS + t_{min}$;
  $P_{ij} \leftarrow ShortestPipeline(CurrentMix)$; $f_{ij} \leftarrow CurrentTS$; $T_{ij} \leftarrow f_{ij} - s_{ij}$;
  Remove P_{ij} from $CurrentMix$;
  foreach $P_{ik} \in CurrentMix$ **do**
    $t^r_{ik} \leftarrow t_{ik} - t_{min}$; // t_{ik} is $\mathcal{M}_{ppl}$'s prediction for P_{ik}
    $\rho^r_{ik} \leftarrow \rho^r_{ik} \cdot \frac{t^r_{ik}}{t_{ik}}$;
  end
  if $HasMorePipelines(\mathcal{P}_i)$ **then**
    Add $P_{i(j+1)}$ into $CurrentMix$; $s_{i(j+1)} \leftarrow CurrentTS$; $\rho^r_{i(j+1)} \leftarrow 1.0$;
  end
  $MakePrediction(CurrentMix, \mathcal{M}_{ppl})$;
  foreach $P_{ik} \in CurrentMix$ **do**
    $t_{ik} \leftarrow \rho^r_{ik} \cdot t_{ik}$;
  end
end

for $1 \leqslant i \leqslant n$ **do**
  $T_i \leftarrow 0$;
  foreach *pipeline* P_{ij} *in* q_i **do**
    $T_i \leftarrow T_i + T_{ij}$;
  end
end
return $\{T_i\}_{i=1}^{n}$;

remaining ratio ρ^r_{ik} (lines 25 to 27). The iteration then repeats by determining the next finishing pipeline.

We call this procedure the *progressive predictor*. The remaining problem is to develop the predictive model $\mathcal{M}_{ppl}$ for a mix of pipelines. We discuss our approaches in Section 3.3.

3.2.4 Analysis

We give some analysis to Algorithm 2, in terms of its efficiency and prediction errors as the number of mixes increases.

3.2.4.1 Efficiency

Whenever $\mathcal{M}_{ppl}$ is called, we must have one pipeline in the current mix that finishes and exits the mix. So the number of times that $\mathcal{M}_{ppl}$ is called cannot exceed the total number of pipelines in the given query mix. Thus we have

Lemma 3.4. *$\mathcal{M}_{ppl}$ is called at most $\sum_{i=1}^{n} |\mathcal{P}_i|$ times, where $\mathcal{P}_i$ is the set of pipelines contained in the query q_i.*

It is possible that several pipelines may finish at the same (predicted) time. In this case, we remove all of them from the current mix, and add each of their successors (if any) into the current mix. We omit this detail in Algorithm 2 for simplicity of exposition. Note that if this happens, the number of times calling $\mathcal{M}_{ppl}$ is fewer than $\sum_{i=1}^{n} |\mathcal{P}_i|$.

3.2.4.2 Prediction Errors

Let the mixes of pipelines in the query mix be M_1, ..., M_n. For the mix M_i, let T_i and T_i' be the *actual* and *predicted* time for M_i. The prediction error D_i is defined as $D_i = \frac{T_i' - T_i}{T_i}$. So $T_i' = T_i(1 + D_i)$. If $D_i > 0$, then $T_i' > T_i$ and it is an overestimation, while if $D_i < 0$, then $T_i' < T_i$ and it is an underestimation. We can view D_1, ..., D_n as i.i.d. random variables with mean μ and variance σ^2. Let D be the overall

prediction error. We have

$$D = \frac{T' - T}{T} = \frac{\sum_{i=1}^{n}(T'_i - T_i)}{T} = \frac{\sum_{i=1}^{n} T_i D_i}{T},$$

where $T = \sum_{i=1}^{n} T_i$ and $T' = \sum_{i=1}^{n} T'_i$, and thus:

Lemma 3.5. $E[D] = \mu$, *and* $Var[D] = \frac{\sum_{i=1}^{n} T_i^2}{\left(\sum_{i=1}^{n} T_i\right)^2}\sigma^2$.

Since $\left(\sum_{i=1}^{n} T_i\right)^2 = \sum_{i=1}^{n} T_i^2 + 2\sum_{1 \leqslant i < j \leqslant n} T_i T_j$, we have $\left(\sum_{i=1}^{n} T_i\right)^2 \geqslant \sum_{i=1}^{n} T_i^2$ and hence $Var[D] \leqslant \sigma^2$, according to Lemma 3.5. This means that the expected overall accuracy is no worse than the expected accuracy of $\mathcal{M}_{ppl}$ over a single mix of pipelines. Intuitively, it is because $\mathcal{M}_{ppl}$ may both overestimate and underestimate some mixes of pipelines, the errors of which are canceled with each other when the overall prediction is computed by summing up the predictions over individual pipeline mixes. So the key to improving the accuracy of the progressive predictor is to improve the accuracy of $\mathcal{M}_{ppl}$.

3.3 Predictive Models

In this section, we present the predictive model $\mathcal{M}_{ppl}$ for a mix of pipelines. $\mathcal{M}_{ppl}$ is based on the cost models used by query optimizers, which basically applies Equation (2.1) to each pipeline. As discussed in Section 3.1, the key challenge is to compute the c's in Equation (2.1) when the pipelines are concurrently running. In the following, we present two alternative approaches. One is a new approach based on previously proposed machine-learning techniques, and the other is a new approach based on analytic models reminiscent of those used by query optimizers. As in previous work [10, 33], we target analytic workloads and assume that queries are primarily I/O-bound.

3.3.1 A Machine-Learning Based Approach

The c's are related to the CPU and I/O interactions between pipelines. These two kinds of interactions are different. CPU interactions are usually *negative*, namely, the pipelines are competing with each other to share CPU cycles. On the other hand, I/O interactions can be either *positive* or *negative* [9] (see Example 3.1 as well). Therefore, we propose separating the modeling of CPU and I/O interactions.

For CPU interactions, we derive a simple model for the CPU-related cost units c_t, c_i, and c_o. For I/O interactions, we extend the idea from [9], using machine-learning techniques to build regression models for the I/O-related cost units c_s and c_r. In the following, we focus on illustrating the basic ideas.

3.3.1.1 Modeling CPU Interactions

We use c_{cpu} to represent c_t, c_i, or c_o. Suppose that we have m CPUs and n pipelines. Let the time to process one CPU request be τ for a standalone pipeline. If $m \geqslant n$, then each pipeline can have its own dedicated CPU, so the CPU time per request for each pipeline is still τ, namely, $c_{cpu} = \tau$. If $m < n$, then we have more pipelines than CPUs. In this case, we assume that the CPU sharing among pipelines is fair, and the CPU time per request for each pipeline is therefore $c_{cpu} = \frac{n}{m}\tau$.

3.3.1.2 Modeling I/O Interactions

Previous work [9] proposed an experiment-driven approach based on machine learning. The idea in that work is the following. Assuming that we know all possible queries (or query types/templates whose instances have very similar execution times) beforehand, we can then run a number of sample mixes of these queries, record their execution time as ground truth, and train a regression model with the data collected. This idea cannot be directly applied to the dynamic workloads we consider here, since it requires prior knowledge of all query templates to be run.

Accordingly, we extend this idea to apply to mixes of pipelines rather than mixes of queries. As a first approximation, we assume the only I/O's performed by a query are due to scans. Later, we relax this assumption. We have the observation:

Observation 1. *For a specific database system implementation, the number of possible* scan *operators is fixed.*

For instance, PostgreSQL implements three scan operators: *sequential scan* (SS), *index scan* (IS), and *bitmap index scan* (BIS).

We define a *scan type* to be a specific scan operator over a specific table. It is then not difficult to see that:

Observation 2. *For a specific database system implementation and a specific database schema, the number of possible scan types is fixed.*

For example, since the TPC-H benchmark database contains 8 tables, and PostgreSQL has 3 scan operators (i.e, SS, IS, and BIS), the total number of possible scan types in this case is then 24.

Using these observations, we can apply the previous machine-learning based approach for scan types instead of query templates. Specifically, in the training stage, we collect sample mixes of scans and build regression models. For each mix of pipelines, we first identify the scans within each pipeline, and then reduce the problem to mixes of scans so that the regression models can be leveraged.

Discussion We assumed for simplicity that the I/O's of a query were only from scans. We now return to this issue. In practice, the I/O's from certain operators (e.g., hash join) due to spilling intermediate results to disk are often not negligible. We have observed in our experiments that completely eliminating these additional I/O's from the model can harm the prediction accuracy by 10% to 30%. Therefore, we choose to incorporate these I/O's into the current model as much as possible. Specifically, we treat the additional I/O's as if they were scans over the underlying tables. For example, PostgreSQL uses the hybrid hash join algorithm. If the partitions produced in the building phase cannot fit in memory, they will be written to disk and read back in the probing phase. This causes additional I/O's. Now suppose that $R \bowtie S$ is a hash join between the table R and S. We model these additional I/O's as additional sequential scans over R and S, respectively.

3.3.2 An Analytic-Model Based Approach

The machine-learning based approach suffers the problem of infinite number of unknown queries. Specifically, the sample space of training data now moves from mixes of queries to mixes of (instance) scans. Note that, although the number of scan types is finite, each scan type can have infinitely many instances. So the number of mixes of instance scans is still infinite. It could be imagined (and also verified in our experimental evaluation) that if the queries contain scans not observed during training, then the prediction is unlikely to be good.

In this section, we present a different approach based on analytic models. Specifically, we model the underlying database system with a queueing network. The c's in Equation (2.1) are equivalent to the *resident times per visit* of the pipelines within the network, and can be computed with standard queueing-network evaluation techniques. Since the queueing network is incapable of characterizing the cache effect of the buffer pool, we further incorporate an analytic model to predict the buffer pool hit rate.

3.3.2.1 The Queueing Network

As shown in Figure 3.5, the queueing network consists of two service centers, one for the disks, and the other for the CPU cores. This is a *closed* model with a *batch* workload (i.e., a *terminal* workload with a think time of zero) [59]. The customers of this queueing system are the pipelines in the mix. In queueing theory terminology, the execution time of a pipeline is its *residence time* in the queueing network.

If both service centers only contain a single server, then it is straightforward to apply the standard mean value analysis (MVA) technique [78] to solve the model. In practice, we usually use the approximate version of MVA for computational efficiency. The results obtained via exact and approximate MVA are close to each other [59]. However, if some service center has multiple servers, the standard technique cannot be directly used, and we instead use the extended approach presented in [86].

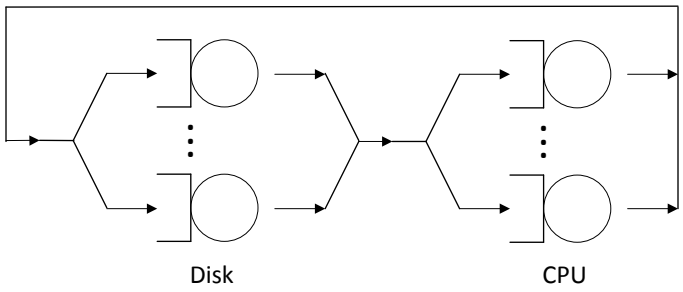


Figure 3.5: A queueing network.

The queueing system shown in Figure 3.5 can be described by the following set of (nonlinear) equations:

$$R_{k,m} = \tau_k + Y_k \tau_k \sum_{j \neq m} Q_{k,j}, \tag{3.1}$$

$$Q_{k,j} = \frac{V_{k,j} R_{k,j}}{\sum_{i=1}^{K} V_{i,j} R_{i,j}}, \tag{3.2}$$

$$Y_k = \frac{1}{C_k} \rho^{4.464(C_k^{0.676}-1)}, \tag{3.3}$$

$$\rho_k = \frac{\tau_k}{C_k} \sum_{j=1}^{M} \frac{V_{k,j}}{\sum_{i=1}^{K} V_{i,j} R_{i,j}}, \tag{3.4}$$

where $k \in \{cpu, disk\}$, and $1 \leqslant m \leqslant M$ (M is the number of customers). Table 3.1 illustrates the notation used in the above equations. Our goal is to compute the residence time $R_{k,m}$ per visit for each customer m at each service center k.

The input parameters of the equations are the τ_k's and $V_{k,m}$'s. τ_k is the mean service time per visit to the service center k. For example, τ_{disk} is the average time for the disk to perform an I/O operation. The τ_k's should be the same as the cost units used for estimating the execution time of a single standalone query. For PostgreSQL, however, we have 5 cost units but we only need 2 τ_k's. We address this issue by picking a *base* cost unit and transform all the other cost units into equivalent amounts of base cost units, with respect to their relative ratios. For example, for the specific machine used in our experiments (see Table 3.3 in Section 3.4), we know that

Notation	Description
C_k	# of servers in (service) center k
τ_k	Mean service time per visit to center k
Y_k	Correction factor of center k
ρ_k	Utility of center k
$V_{k,m}$	Mean # of visits by customer m to center k
$Q_{k,m}$	Mean queue length by customer m at center k
$R_{k,m}$	Mean residence time per visit by customer m to center k

Table 3.1: Notation used in the queueing model.

$c_r = 11.3c_s$, which means the time of 1 random I/O is equivalent to 11.3 sequential I/O's. In our experiments, we pick $\tau_{disk} = c_r$ and $\tau_{cpu} = c_t$ as the base I/O and CPU cost unit (the other choices are also OK). Then the number of I/O and CPU visits $V_{k,m}$ of a pipeline are $(n_r + n_s \cdot \frac{c_s}{c_r})$ and $(n_t + n_i \cdot \frac{c_i}{c_t} + n_o \cdot \frac{c_o}{c_t})$. The n's of a pipeline are computed based on the n's of each operator in the pipeline. Specifically, suppose that a pipeline contains l operators O_1, ..., O_l. Let n_j (n_j can be any of the n_s, n_r, etc) be the optimizer's estimate for the operator O_j. The corresponding quantity for the pipeline is then $\sum_{j=1}^{l} n_j$.

If there is only one server in the service center k (i.e., $C_k = 1$), then $Y_k = 1$ by Equation (3.3). Equation (3.1) is then reduced to the case of standard MVA, which basically says that the residence time $R_{k,m}$ is sum of the service time τ_k and the queueing time $\tau_k \sum_{j \neq m} Q_{k,j}$. The expression of the queueing time is intuitively the sum of the queueing time of the customers other than the customer m, each of which in turn is the product of the queue length for each class (i.e., $Q_{k,j}$) and their service time (i.e., τ_k).

When there are multiple servers in the service center, intuitively the queueing time would be less than if there were only one server. The *correction factor* Y_k is introduced for this purpose. The formula of Y_k in Equation (3.3) was derived in [86], and was shown to be good in their simulation results.

By substituting Equation (3.2) to (3.4) into Equation (3.1), we can obtain a system of nonlinear equations where the only unknowns are the $R_{k,m}$'s. We use the `fsolve` function of Scilab [81] to solve this system. Any other equivalent solver can be used for this purpose as well.

3.3.2.2 The Buffer Pool Model

The weakness of the queueing network introduced above is that it does not consider the effect of the buffer pool. Actually, since the buffer pool plays the role of eliminating I/O's, it cannot be viewed as a service center and therefore cannot be modeled within the queueing network. We hence need a special-purpose model here to predict the buffer pool hit rate. Of course, different buffer pool replacement policies need different models. We adapt the analytic model introduced in [70] for the "clock" algorithm that is used in PostgreSQL. If a system uses a different algorithm (e.g., LRU, LRU-k, etc), a different model should be used.

The clock algorithm works as follows. The pages in the buffer pool are organized in a circular queue. Each buffer page has a counter that is set to its maximum value when the page is brought into the buffer pool. On a buffer miss, if the requested page is not in the buffer pool and there is no free page in the buffer, a current buffer page must be selected for replacement. The clock pointer scans the pages to look for a victim. If a page has count 0, then this page is chosen for replacement. If a page has a count larger than 0, then the count is decreased by 1 and the search proceeds. On a buffer hit, the counter of the page is reset to its maximum value.

The analytic approach in [70] models this procedure by using a Markov chain. Suppose that we have P partitions in the system (we will discuss the notion of partition later). Let h_p be the buffer pool hit rate for the partition p, where $1 \leqslant p \leqslant P$. h_p can be obtained by solving the following system of equations:

$$\sum_{p=1}^{P} S_p\big(1 - \frac{1}{(1 + \frac{n_0}{m}\frac{r_p}{S_p})^{I_p+1}}\big) - B = 0, \tag{3.5}$$

$$N_p = S_p\big(1 - \frac{1}{(1 + \frac{n_0}{m}\frac{r_p}{S_p})^{I_p+1}}\big), \tag{3.6}$$

$$h_p = \frac{N_p}{S_p}. \tag{3.7}$$

The notation used in the above equations is illustrated in Table 3.2.

Notation	Description
n_0	Mean # of buffer pages with count 0
m	Overall buffer pool miss rate
S_p	# of pages in partition p
r_p	Probability of accessing partition p
I_p	Maximum value of the counter of partition p
N_p	Mean # of buffer pool pages from partition p
h_p	Buffer pool hit rate of partition p

Table 3.2: Notation used in the buffer pool model.

By Equation (3.6) and (3.7),

$$m_p = 1 - h_p = [(1 + \frac{n_0}{m}\frac{r_p}{S_p})^{I_p+1}]^{-1}$$

represents the buffer *miss* rate of the partition p. Note that n_0 can be thought of as the number of buffer misses that can be handled in one clock cycle. As a result, $\frac{n_0}{m}$ is the number of buffer accesses (including both buffer hits and misses) in one clock cycle. Hence $\frac{n_0}{m}\frac{r_p}{S_p}$ is the expected number of accesses to a page in the partition p. Intuitively, the higher this number is, the more likely the page is in the buffer pool and hence the smaller m_p is. The expression of m_p captures this intuition.

It is easy to see that we can determine the quantity $\frac{n_0}{m}$ from Equation (3.5), since it is the only unknown there. We can then figure out N_p and hence h_p by examining Equation (3.6) and Equation (3.7). To solve $\frac{n_0}{m}$ from Equation (3.5), define

$$F(t) = \sum_{p=1}^{P} S_p\big(1 - \frac{1}{(1 + t \cdot \frac{r_p}{S_p})^{I_p+1}}\big) - B.$$

We have $F(0) = -B < 0$, and $F(+\infty) = \lim_{t\to+\infty} F(t) = \big(\sum_{p=1}^{P} S_p\big) - B > 0$, since we except the size of the database $\sum_{p=1}^{P} S_p$ is bigger than the size of the buffer pool B (in pages). Since $F(t)$ is strictly increasing as t increases, we know that there is some $t_0 \in [0, +\infty)$ such that $F(t_0) = 0$. We can then use a simple but very efficient bisection method to find t_0 [70]. Here, B, $\{S_p\}_{p=1}^{P}$, and $\{I_p\}_{p=1}^{P}$ are measurable system

parameters. $\{r_p\}_{p=1}^P$ can be computed based on $\{S_p\}_{p=1}^P$ and the number of I/Os to each partition, which can be obtained from the query plans.

The remaining issue is how to partition the database. The partitioning should not be arbitrary because the analytic model is derived under the assumption that the access to database pages within a partition is uniform. An accurate partitioning thus requires information about access frequency of each page in the database, which depends on the particular workload to the system. For the TPC-H workload we used in our experiments, since the query templates are designed in some way that a randomly generated query instance is equally likely to touch each page,[1] we simplified the partitioning procedure by treating each TPC-H table as a partition. In a real deployed system, we can further refine the partitioning by monitoring the access patterns of the workload [70].

3.3.2.3 Putting It Together

The complete predictive approach based on the analytic models is summarized in Algorithm 3. We first call the analytic model $\mathcal{M}_{buf}$ to make a prediction for the buffer pool hit rate h_p of each partition p (line 1). Since only buffer pool misses will cause actual disk I/O's, we discount the disk visits $V_{disk,i,p}$ of each partition p accessed by the pipeline i with the buffer pool miss rate $(1 - h_p)$. The disk visits $V_{disk,i}$ of the pipeline i is the sum of its visits to each partition (lines 2 to 7). We then call the queueing model $\mathcal{M}_{queue}$ to make a prediction for the residence time per visit of the pipeline i in the service center k, where $k \in \{cpu, disk\}$ (line 8). The predicted execution time T_i for the pipeline i is simply (line 10):

$$T_i = V_{cpu,i} R_{cpu,i} + V_{disk,i} R_{disk,i}.$$

It might be worth noting that, the queueing model here is equivalent to the optimizer's cost model when there is only one single pipeline. To see this, notice

[1] Specifically, the TPC-H benchmark database is uniformly generated. The TPC-H queries usually use pure sequential scans or index scans with range predicates to access the tables. If it is a sequential scan, then clearly the access to the table pages is uniform. If it is an index scan, the range predicate is uniformly generated so that each page in the table is equally likely to be touched.

Algorithm 3: $\mathcal{M}_{ppl}$ based on analytic models.

Input: $\{P_1, ..., P_n\}$, a mix of n pipelines; $\mathcal{M}_{queue}$: the queueing model; $\mathcal{M}_{buf}$: the buffer pool model

Output: $\{T_i\}_{i=1}^{n}$, where T_i is the predicted execution time of the pipeline P_i

```
1  {h_p}_{p=1}^{P} ← PredictHitRate(M_buf);
2  for 1 ⩽ i ⩽ n do
3  |  V_{disk,i} ← 0;
4  |  foreach partition p accessed by P_i do
5  |  |  V_{disk,i} ← V_{disk,i} + V_{disk,i,p}(1 − h_p);
6  |  end
7  end
8  {R_{k,i}}_{i=1}^{n} ← PredictResTime(M_queue, {V_{k,i}}_{i=1}^{n});
9  for 1 ⩽ i ⩽ n do
10 |  T_i ← V_{cpu,i}R_{cpu,i} + V_{disk,i}R_{disk,i};
11 end
12 return {T_i}_{i=1}^{n};
```

that the $\sum_{j \neq m} Q_{k,j}$ in the second summand of Equation (3.1) vanishes if there is only one customer. Therefore, we simply have $R_{k,m} = \tau_k$ in this case. Due to the use of base cost units, no information is lost when multiplying $V_{k,m}$ by τ_k. Specifically, for example, suppose that $k = disk$. We have

$$V_{disk,m} \cdot \tau_{disk} = (n_r + n_s \cdot \frac{c_s}{c_r}) \cdot c_r = n_r \cdot c_r + n_s \cdot c_s,$$

which is the same as the optimizer's estimate. Since the progressive predictor degenerates to summing up the predicted time of each individual pipeline if there is only one query, the predicted execution time of the query is therefore the same as what if the optimizer's cost model is used. In this regard, for single-query execution time prediction, the analytic-model based approach here can also be viewed as a new predictor based on the optimizer's cost model, with the addition of the buffer pool model that further predicts buffer pool hitting rate.

3.4 Experimental Evaluation

In this section, we present our experimental evaluation of the proposed approaches. We measure the prediction accuracy in terms of *mean relative error* (MRE), a metric used in [10, 33]. MRE is defined as

$$\frac{1}{N}\sum_{i=1}^{N}\frac{|T_i^{pred} - T_i^{act}|}{T_i^{act}}.$$

Here N is the number of testing queries, T_i^{pred} and T_i^{act} are the predicted and actual execution times for query i. We measured the overhead of the prediction approaches as well.

3.4.1 Experimental Setup

We evaluated our approaches with the TPC-H 10GB benchmark database. In our experiments, we varied the multiprogramming level (MPL), i.e., the number of queries that were concurrently running, from 2 to 5. All the experiments were conducted on a machine with dual Intel 1.86 GHz CPU and 4GB of memory. We ran PostgreSQL 9.0.4 under Linux 3.2.0-26.

3.4.1.1 Workloads

We used the following two TPC-H-based workloads and three micro-benchmarking workloads in our experiments:

I. TPC-H workloads

• **TPC-H1**: This is a workload created with 9 TPC-H query templates that are of light to moderate weight queries. Specifically, the templates we used are TPC-H queries 1, 3, 5, 6, 10, 12, 13, 14, and 19. We choose light to moderate queries because they allow us to explore higher MPL's without overloading the system [33]. For each MPL, we then generated mixes of TPC-H queries via Latin Hypercube Sampling

(LHS) [10, 33]. LHS creates a hypercube with the same dimensionality as the given MPL. Each dimension is divided into T equally probable intervals marked with 1, 2, ..., T, where T is the number of templates. The interval i represents instances of the template i. LHS then selects T sample mixes such that every value in every dimension appears in exact one mix. Intuitively, LHS has better coverage of the space of mixes than uniformly random sampling, given that the same number of samples are selected. The purpose of TPC-H1 is to compare different approaches over uniformly generated query mixes.

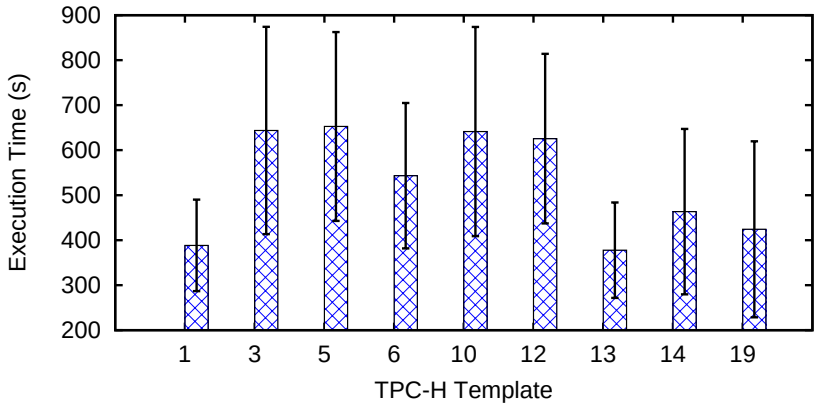
• **TPC-H2**: This workload is generated in the same way as TPC-H1. In addition to the 9 templates in TPC-H1, we added 3 more expensive TPC-H templates 7, 8, and 9. The purpose is to test the approaches under a more *diverse* workload, in terms of the distribution of query execution times. Figure 3.6 compares the variance in query execution times of TPC-H1 and TPC-H2, by presenting the mean and standard deviation (shown as error bars) of the execution times of queries in each TPC-H template. As we can see, the execution times of some queries (e.g., Q3 and Q5) are much longer in TPC-H2 than in TPC-H1, perhaps due to the more severe interactions with the three newly-added, long-running templates.

II. Micro-benchmarking workloads

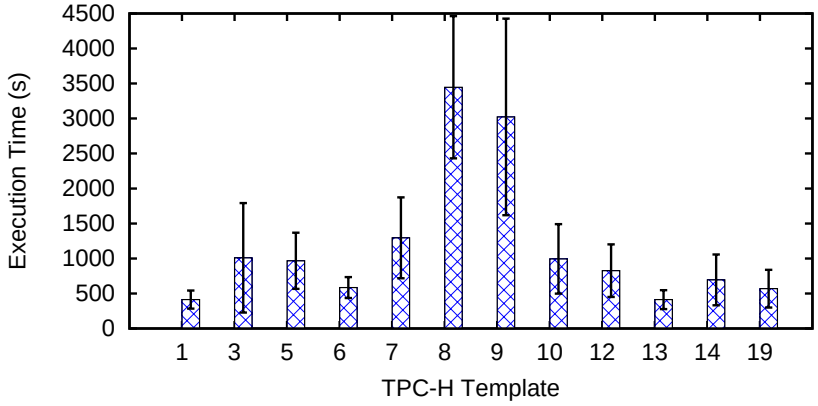
• **MB1**: This is a workload with 36 mixes of queries, 9 for each MPL from 2 to 5. A mix for MPL m contains m queries of the following form:

```
SELECT * FROM lineitem
WHERE l_partkey > a and l_partkey < b.
```

Here *l_partkey* is an attribute of the *lineitem* table with an unclustered index. The values of *l_partkey* is between 0 and 2,000,000. We vary a and b to produce index scans with data sharing ratio 0, 0.1, ..., 0.8. For example, when MPL is 2, if the data sharing ratio is 0, the first scan is generated with $a = 0$ and $b = 1,000,000$, and the second scan is generated with $a = 1,000,000$ and $b = 2,000,000$; if the data sharing ratio is



(a) TPC-H1



(b) TPC-H2

Figure 3.6: Variance in query execution times.

0.2, then the first scan is generated with $a = 0$ and $b = 1,111,111$, while the second scan is generated with $a = 888,888$ and $b = 2,000,000$. The purpose of MB1 is to compare different approaches over query mixes with different data sharing ratios.

• **MB2**: This is a workload with mixes that mingle both sequential and index scans. We focus on the two biggest tables *lineitem* and *orders*. For each table, we include 1 sequential scan and 5 index scans, and there is no data sharing between the index scans. For each MPL from 2 to 5, we generate query mixes by enumerating all pos-

sible combinations of scans. For example, when MPL is 2, we can have 10 different mixes, such as 2 sequential scans over *lineitem*, 1 sequential scan over *lineitem* and 1 sequential scan over *orders*, and so on. Whenever an index scan is required, we randomly pick one from the five candidates. The purpose of MB2 is to compare different approaches over query mixes with different proportion of sequential and random accesses.

• **MB3**: This is a workload similar to MB2, for which we replace the scans in MB2 with TPC-H queries. We do this by classifying the TPC-H templates based on their scans over the *lineitem* and *orders* table. For example, the TPC-H Q1 contains a sequential scan over *lineitem*, and Q13 contains a sequential scan over *orders*. When generating a query mix, we first randomly pick a TPC-H template containing the required scan, and then randomly pick a TPC-H query instance from that template. The purpose of MB3 is to repeat the experiments on MB2 with less artificial, more realistic query mixes.

3.4.1.2 Calibrating PostgreSQL's Cost Models

Optimizer Parameter	Calibrated μ (ms)	Default
seq_page_cost (c_s)	8.52e-2	1.0
random_page_cost (c_r)	9.61e-1	4.0
cpu_tuple_cost (c_t)	2.04e-4	0.01
cpu_index_tuple_cost (c_i)	1.07e-4	0.005
cpu_operator_cost (c_o)	1.41e-4	0.0025

Table 3.3: Actual values of PostgreSQL optimizer parameters.

Both the machine-learning and analytic-model based approaches need the c's and n's from the query plan as input. However, as we have discussed in Chapter 2, the crude values of these quantities might be incorrect and hence are not ready for use. We therefore use the framework proposed in Chapter 2 to calibrate the c's and refine the n's. Table 3.3 presents the calibrated values for the 5 cost units on the machine used in our experiments. For the n's, we use the aforementioned

sampling-based method by setting the sampling ratio to be 0.05 (i.e., the sample size is 5% of the database size). The prediction accuracy of our proposed approaches observed on the tested workloads using this sampling ratio is quite close to that observed using the *true* cardinalities to compute the n's.

3.4.1.3 Settings for Machine Learning

As mentioned before, the TPC-H benchmark database consists of 8 tables, 6 of which have indexes. Also, there are 3 kinds of scan operators implemented by PostgreSQL, namely, sequential scan (SS), index scan (IS), and bitmap index scan (BIS). Therefore, we have 8 SS scan types, one for each table, and 6 IS scan types, one for each table with some index. Since BIS's are rare, we focus on the two biggest tables *lineitem* and *orders* for which we observed the occurrences of BIS's in the query plans. By including these 2 BIS scan types, we have 16 scan types in total. We then use Latin Hypercube Sampling (LHS) to generate sample mixes of scan types. For a given sample mix, we further randomly generate an instance scan for each scan type in the mix. Since we have 16 scan types, each run of LHS can generate 16 sample mixes. While we can run LHS many times, executing these mixes to collect training data is costly. Hence, for each MPL, we run LHS 10 times.

ID	Description
F1	# of sequential I/O's of s_i
F2	# of random I/O's of s_i
F3	# of scans in $N(s_i)$ that are over tbl_i
F4	# of sequential I/O's from scans in $N(s_i)$ that are over tbl_i
F5	# of random I/O's from scans in $N(s_i)$ that are over tbl_i
F6	# of scans in $N(s_i)$ that are *not* over tbl_i
F7	# of sequential I/O's from scans in $N(s_i)$ that are *not* over tbl_i
F8	# of random I/O's from scans in $N(s_i)$ that are *not* over tbl_i

Table 3.4: Features of s_i.

We used the features in Table 3.4 to represent an scan instance s_i in a mix $\{s_1, ..., s_n\}$, where tbl_i is the table accessed by s_i, and $N(s_i)$ is the set of neighbor scans of s_i in the mix. We tested representatives of both linear models and nonlinear

models. For linear models, we used multivariate linear regression (MLR), and for nonlinear models, we used REP regression trees (REP) [74]. We also tested the well-known boosting technique that combines predictions from multiple models, which is generally believed to be better than a single model. Specifically, we used additive regression [35] here, with shallow REP trees as base learners. All of these models can be obtained from the WEKA software package [46]. In our study, we found that REP trees outperformed both linear regression and additive regression, in terms of prediction accuracy. Therefore, in the following we only present the results of the machine-learning based approach by using the REP trees.

3.4.1.4 Settings for Analytic Models

The queueing model needs the calibrated c's (in Table 3.3) and n's as input. In addition, the buffer pool model also requires a dozen parameters. Table 3.5 lists the values of these parameters for the system and database configurations used in our experiments.

Parameter	Description	Value
B	# of buffer pool pages	439,463
I_p	Max counter value (for all p)	5
$S_{lineitem}$	# of pages in *lineitem*	1,065,410
S_{orders}	# of pages in *orders*	253,278
$S_{partsupp}$	# of pages in *partsupp*	170,916
S_{part}	# of pages in *part*	40,627
$S_{customer}$	# of pages in *customer*	35,284
$S_{supplier}$	# of pages in *supplier*	2,180
S_{nation}	# of pages in *nation*	1
S_{region}	# of pages in *region*	1

Table 3.5: Values of buffer pool model parameters.

3.4.2 Prediction Accuracy

We evaluated the accuracy of our approaches with the five workloads described in Section 3.4.1.1. To see the effectiveness of our approaches, in our evaluation we

also included a simple baseline approach:

Baseline: For each query in the mix, predict its execution time as if it were the only query running in the database system, by using the method described in Chapter 2. Then multiply it with the MPL (i.e., the number of queries in the mix) as the prediction for the query. Intuitively, this approach ignores the impact of interactions from different neighbors of the query. It will produce the same prediction for the query as long as the MPL is not changed.

3.4.2.1 Results on TPC-H Workloads

Figure 3.7 and 3.8 present the prediction errors over the two TPC-H-based workloads TPC-H1 and TPC-H2. On TPC-H1, the accuracy of the analytic-model based and the machine-learning based approach are close, both outperforming the baseline approach by reducing the error by 15% to 30% (Figure 3.7). This performance improvement may not be very impressive in view of the simplicity of the baseline approach. However, we note here that this is because of the way the workload is generated rather than a problem in our approaches. The workload TPC-H1 turns out to be relatively easy to predict (the errors of all approaches are relatively small). When we move to the more diverse workload TPC-H2, the prediction accuracy of the baseline approach deteriorates dramatically (Figure 3.8), while its two competitors retain prediction accuracy close to that observed on TPC-H1. Nonetheless, it is important to include the baseline to show that sometimes it does surprisingly well, and makes it challenging to improve substantially over that baseline. We also observe that, on TPC-H2, the analytic-model based approach slightly outperforms the machine-learning based approach, improving the accuracy by about 10%.

3.4.2.2 Results on Micro-Benchmarking Workloads

Since the TPC-H workloads were generated via LHS, they cover only a small fraction of the space of possible query mixes. As a result, many particular kinds of query interactions might not be captured. We therefore evaluated the proposed

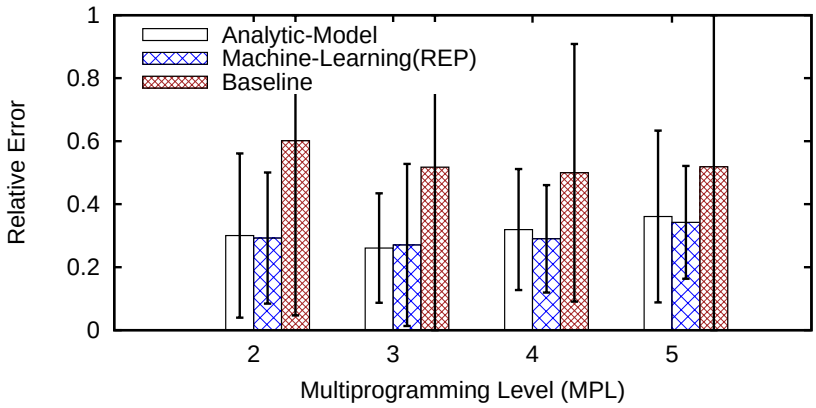


Figure 3.7: Prediction error on TPC-H1 for different approaches.

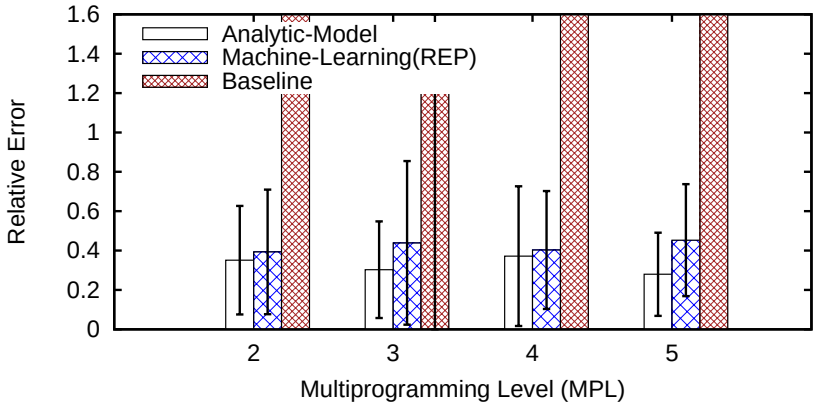


Figure 3.8: Prediction error on TPC-H2 for different approaches.

approaches over the three micro-benchmarking workloads as well, which were more diverse than the TPC-H workloads in terms of query interactions. Figure 3.9 to 3.11 present the results.

On MB1, the prediction errors of the machine-learning based and the baseline approach are very large, while the errors of the analytic-model based approach remain small (Figure 3.9). The baseline approach fails perhaps because it does not take the data sharing between queries into consideration. We observed consistent overestimation made by the baseline approach, while the analytic-model based approach correctly detected the data sharing and hence leveraged it in buffer pool

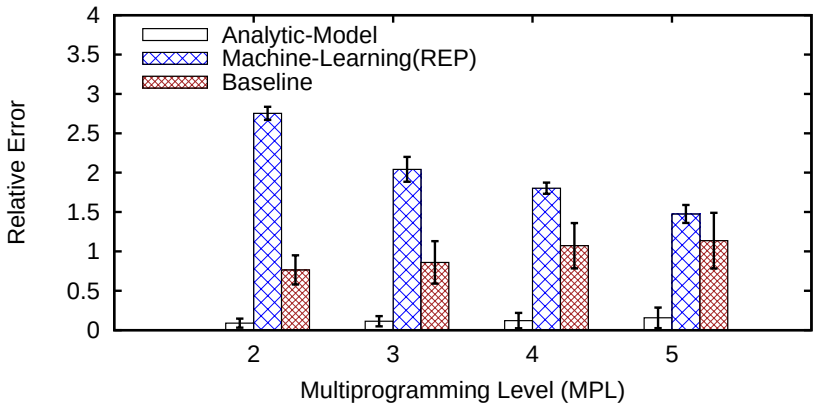


Figure 3.9: Prediction error on MB1 for different approaches.

hit rate prediction. The machine-learning based approach is even worse than the baseline approach. This is because we train the model with mixes of scans generated via LHS, which are quite different from the mixes of scans in MB1. MB1 focuses on heavy index scans over a particular table. In typical LHS runs, very few samples can be obtained from such a specific region since the goal of LHS is to uniformly cover the whole huge space of query mixes.

The prediction errors of the baseline approach remain large on the workloads MB2 and MB3 (Figure 3.10 and 3.11). This is not surprising, since the query interactions in MB2 and MB3 are expected to be much more diverse than they are in the TPC-H workloads. It is hard to believe that a model ignoring all these interactions can work for these workloads. Meanwhile, the analytic-model based approach is still better than the machine-learning based approach on MB2, by reducing the prediction errors by 20% to 25%, and they are comparable on MB3. One possible reason for this improvement of the machine-learning based approach may be that the interactions in MB2 and MB3 are closer to what it learnt during training. Recall that we intentionally enforce no data sharing among the index scans used to generate MB2 and MB3, and hence the index scans are somewhat independent of each other. This is similar to what LHS did in training, for which the scans in a mix are independently generated. This is quite different for MB1, however, where the queries are correlated due to data sharing.

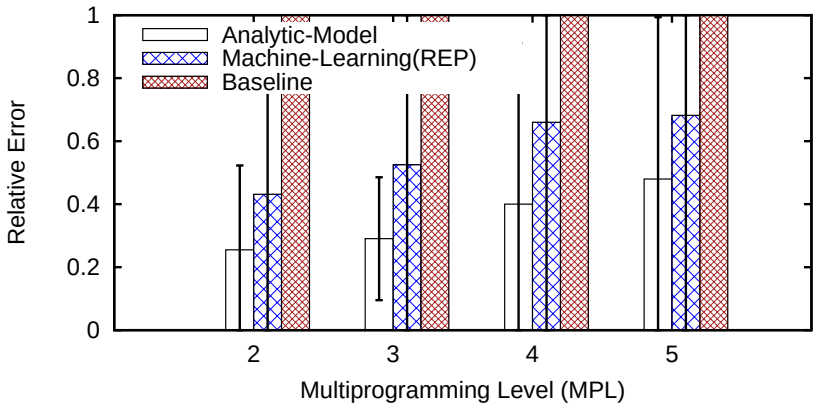


Figure 3.10: Prediction error on MB2 for different approaches.

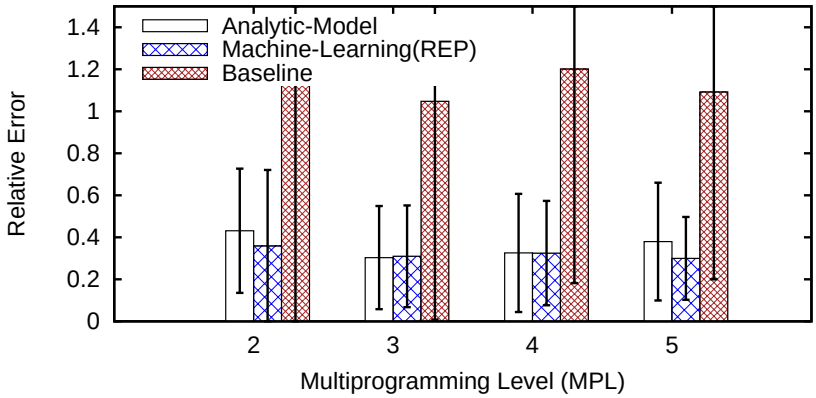


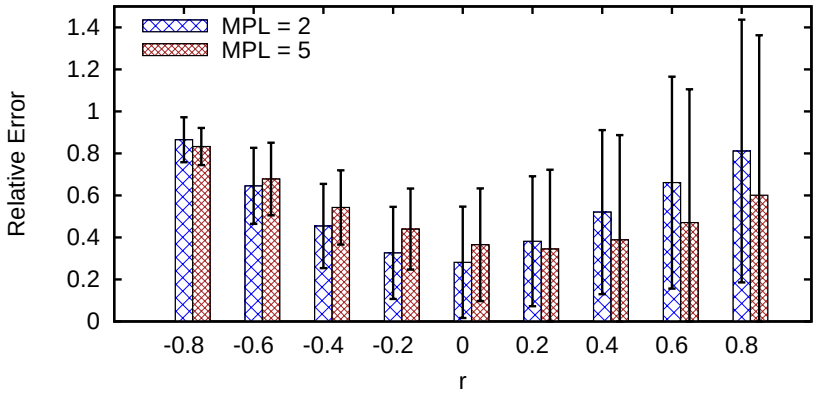
Figure 3.11: Prediction error on MB3 for different approaches.

3.4.2.3 Sensitivity to Errors in Cardinality Estimates

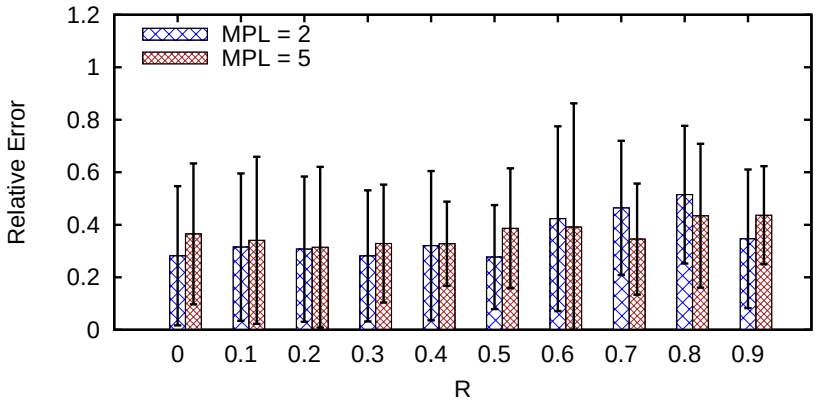
Both the machine-learning based and the analytic-model based approach rely on the π's from the query plans. Since the accuracy of the π's depends on the quality of cardinality estimates, which are often erroneous in practice, a natural question is how sensitive the proposed approaches are to errors in cardinality estimates.

We investigated this question for the analytic-model based approach, which, as we have seen, outperformed its machine-learning counterpart on the workloads we tested. We studied this by feeding the optimizer cardinalities generated by perturbing the *true* cardinalities. Specifically, consider an operator O with true

input cardinality N_O. Let r be the *error rate*. In our perturbation experiments, instead of using N_O, we used $N'_O = N_O(1 + r)$ to compute the π's of O. We considered both *biased* and *unbiased* errors. The errors are biased if we use the same error rate r for all operators in the query plan, and the errors are unbiased if each operator uniformly randomly draws r from some interval $(-R, R)$.



(a) Biased errors



(b) Unbiased errors

Figure 3.12: Sensitivity of prediction accuracy on TPC-H1.

Figure 3.12 shows the results on the TPC-H1 workload. We observe that in the presence of biased errors, the prediction errors increase in proportion to the errors in cardinality estimates. However, the prediction errors often increase more

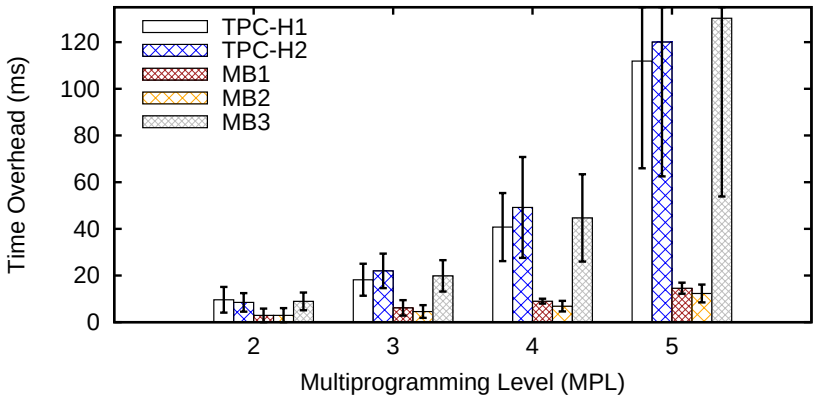


Figure 3.13: Runtime overhead in evaluating analytic models.

slowly than the cardinality estimation errors. For example, in Figure 3.12(a), the mean prediction error increases from 0.36 to 0.47 for MPL 5 when r increases from 0 to 0.6. On the other hand, the prediction accuracy is more stable in the presence of unbiased errors. As shown in Figure 3.12(b), the prediction errors are almost unchanged when R increases from 0 to 0.4. The intuition for this is that if the errors are unbiased, then for each operator in the query plan, it is equally likely to overestimate or underestimate its cardinalities. Therefore, the errors might cancel each other when making prediction for the entire query.

3.4.3 Additional Overhead

Both the machine-learning based and the analytic-model based approach need to calibrate the optimizer's cost model. As discussed in Chapter 2, calibrating the c's is a one-time procedure and usually can be done within a couple of hours. The overhead of calibrating the n's via sampling depends on the sample size. For the sampling ratio 0.05, it takes around 4% of the query execution time, when the samples are disk-resident. This overhead could be substantially reduced using the common techniques of keeping the samples in memory [75].

In addition to the overhead in calibrating the cost model, the machine-learning based approach needs to collect the training data. Although the training is offline,

this overhead is not trivial. The time spent in the training stage depends on several factors, such as the number of sample scan mixes and the overhead of each scan instance. For the specific settings used in our experiments, the training stage usually takes around 2 days.

On the other hand, the analytic-model based approach needs to evaluate the analytic models when making the prediction. This includes the time of solving the systems of nonlinear equations required by both the queueing model and the buffer pool model. Figure 3.13 presents the average total time spent in the evaluation as well as the standard deviation (as error bars). As expected, the time overhead increases as the MPL grows, since the queueing model becomes more complicated. However, the overall time overhead is ignorable (e.g., around 120 ms when MPL is 5), compared with the execution time of the queries (usually hundreds of seconds).

3.4.4 Discussion

The approaches proposed in this chapter relies on the optimizer's cost model to provide reasonable cost estimates. Although we have used the framework in Chapter 2 to calibrate the cost model, it still contains some flaws. For example, in the current implementation of PostgreSQL, the cost model does not consider the heterogeneous resource usage at different stages of an operator. This may cause some inaccuracy in the cost estimates. For instance, the I/O cost per page in the building phase of hash-based joins might be longer than that predicted by the cost model, due to potential read/write interleaves if spilling occurs. In this case, the current approach might underestimate the execution time of a hash join operator. One way to fix these issues is to improve the cost model.

For example, in [72], the authors proposed a more accurate analytic model for the hybrid hash join algorithm, by further considering the read/write interleavings in the building phase. A thorough revision to the PostgreSQL's cost model, however, might require considerable development efforts and is beyond the scope of this chapter. Our goal here is just to see how effective the proposed approach is, based on the currently-used imperfect cost models. We believe that an improved cost

model could further enhance our approach by delivering more accurate predictions, and we leave the development of a better cost model as interesting future work.

3.5 Related Work

The problem of predicting concurrent query execution time was studied in [10] and [33]. In [10], the authors proposed an experiment-driven approach by sampling the space of possible query mixes and fitting statistical models to the observed execution time of these samples. Specifically, they used Gaussian processes as the particular statistical model. A similar idea was used in [33], where the authors proposed predicting the buffer access latency (BAL) of a query, which is the average delay between the time when an I/O request is issued and the time when the requested block is returned. BAL was found to be highly correlated with the query execution time, and they simply used linear regression mapping BAL to the execution time. To predict BAL, the authors collected training data by measuring the BALs under different query mixes and then built a predictive model based on multivariate regression. The key limitation of both work is that they both assumed static workloads, which is usually not the case in practice. To the best of our knowledge, we are the first that addresses the concurrent query execution time prediction problem under dynamic workloads.

Queueing networks have been extensively used in computer system modeling, including database systems (e.g., [71, 83, 87]). However, the focus in this work is quite different from ours. Previous work used queueing networks to predict macro performance metrics such as the throughput and mean response time for different workloads. Their goal, as pointed out by Sevcik [83], was "predicting the *direction* and approximate *magnitude* of the change in performance caused by a particular design modification." As a result, the models were useful as long as they could correctly predict the trend in system performance, although "significant errors in absolute predictions of performance" were possible. In contrast, our goal is to predict the exact execution time for each individual query. Due to this discrepancy, we applied queueing networks in a quite different manner. Previous

work modeled the system as an open network (e.g., [71, 87]), the evaluation of which heavily relies on assumptions about query arrival rates and service time distributions (e.g., M/M/1 and M/G/1 queues). In contrast, we do not assume any additional workload knowledge except for the current query mix to be predicted, since we target dynamic workloads. Therefore, we modeled the system as a closed network, and used the mean value analysis (MVA) technique to solve the model. Moreover, we treated pipelines rather than the entire queries as customers of the queueing network, motivated by the observation that query interactions happen at the pipeline level rather than at the query level.

3.6 Summary

In this chapter, we studied the problem of predicting query execution time for concurrent and dynamic database workloads. Our approach is based on analytic models, for which we first use the optimizer's cost model to estimate the I/O and CPU operations for each individual query, and then use a queueing model to combine these estimates for concurrent queries to predict their execution times. A buffer pool model is also incorporated to account for the cache effect of the buffer pool. We show that our approach is competitive to and often better than a variant of previous machine-learning based approaches, in terms of prediction accuracy.

We regard this chapter as a first step towards this important but challenging problem. To improve the prediction accuracy, one could either try new machine-learning techniques or develop better analytic models. While previous work favored the former option, the results shown in this chapter shed some light on the latter one. Moreover, a hybrid approach combining the merits of both approaches is worth consideration for practical concern, since most database workloads are neither *purely* static nor *purely* dynamic. All these directions deserve future research effort.

Chapter 4

Uncertainty-Aware Query Execution Time Prediction

In Chapter 2 and 3, we have discussed the problem of predicting query execution time for both single-query and multi-query workloads, which is a fundamental issue underlying many database management tasks. As we have seen, existing predictors rely on information such as cardinality estimates and system performance constants that are difficult to know exactly. As a result, accurate prediction still remains elusive for many queries. However, existing predictors (including ours) provide a single, point estimate of the true execution time, but fail to characterize the uncertainty in the prediction.

In this chapter, we take a first step towards providing uncertainty information along with query execution time predictions. As before, we use the query optimizer's cost model to represent the query execution time as a function of the selectivities of operators in the query plan as well as the constants that describe the cost of CPU and I/O operations in the system. By treating these quantities as random variables rather than constants, we show that with low overhead we can infer the distribution of likely prediction errors. We further show that the estimated prediction errors by our proposed techniques are strongly correlated with the actual prediction errors.

4.1 Introduction

It is a general principle that if there is uncertainty in the estimate of a quantity, systems or individuals using the estimate can benefit from information about this uncertainty. (As a simple but ubiquitous example, opinion polls cannot be reliably interpreted without considering the uncertainty bounds on their results.) In view of this, it is somewhat surprising that something as foundational as query running time estimation typically does not provide any information about the uncertainty embedded in the estimates.

There is already some early work indicating that providing this uncertainty information could be useful. For example, in approximate query answering [48, 55], approximate query results are accompanied by error bars to indicate the confidence in the estimates. It stands to reason that other user-facing running time estimation tasks, for example, query progress indicators [23, 64], could also benefit from similar mechanisms regarding uncertainty. Other examples include robust query processing and optimization techniques (e.g., [14, 28, 39, 40, 66, 90]) and distribution-based query schedulers [26]. We suspect that if uncertainty information were widely available many more applications would emerge.

In this chapter, we study the problem of providing uncertainty information along with query execution time predictions. In particular, rather than just reporting a point estimate, we provide a distribution of likely running times. There is a subtlety in semantics involved here — the issue is not "if we run this query 100 times what do we think the distribution of running times will be?" Rather, we are reporting "what are the likelihoods that the actual running time of this query would fall into certain confidence intervals?" As a concrete example, the distribution conveys information such as "I believe, with probability 70%, the running time of this query should be between 10s and 20s."

Following our discussion in Chapter 2, we use query optimizers' cost models to represent the query execution time as a function of selectivities of operators in the query plan as well as basic system performance parameters such as the unit cost of a single CPU or I/O operation (i.e., cost units). However, our approach here is

different from that in Chapter 2 — we treat these quantities as random variables rather than fixed constants. We then use sampling based approaches to estimate the distributions of these random variables. Based on that, we further develop analytic techniques to infer the distribution of likely running times.

In more detail, for specificity consider again the cost model used by the query optimizer of PostgreSQL as described in Example 2.1, where the execution runtime overhead t_O of an operator O (e.g., scan, sort, join, etc.) is estimated as:

$$t_O = n_s \cdot c_s + n_r \cdot c_r + n_t \cdot c_t + n_i \cdot c_i + n_o \cdot c_o. \tag{4.1}$$

Here the c's are *cost units* as described in Example 2.1. The total estimated overhead t_q of a query q is simply the sum of the costs of the individual operators in its query plan. Moreover, as we discussed, the n's are actually functions of the input/output cardinalities (or equivalently, selectivities) of the operators. As a result, we can further represent t_q as a function of the cost units $\mathbf{c}$ and the selectivities $\mathbf{X}$, namely,

$$t_q = \sum_{O \in Plan(q)} t_O = g(\mathbf{c}, \mathbf{X}). \tag{4.2}$$

Perfect predictions therefore rely on three assumptions: (i) the c's are accurate; (ii) the X's are accurate; and (iii) g is itself accurate. Unfortunately, none of these holds in practice. First, the c's are inherently random. For example, the value of c_r may vary for different disk pages accessed by a query, depending on where the pages are located on disk. Second, accurate selectivity estimation is often challenging, though significant progress has been made. Third, the equations and functions modeling query execution make approximations and simplifications so they could make errors. For instance, Equation (4.1) does not consider the possible interleaving of CPU and I/O operations during runtime.

To quantify the uncertainty in the prediction, we therefore need to consider potential errors in all three parts of the running time estimation formula. It turns out that the errors in the c's, the X's, and g are inherently different. The errors in the c's result from fluctuations in the system state and/or variances in the way

the system performs for different parts of different queries. (That is, for example, the cost of a random I/O may differ substantially from operator to operator and from query to query.) We therefore model the c's as random variables and extend our calibration framework in Chapter 2 to obtain their distributions. The errors in the X's arise from selectivity estimation errors. We therefore also model these as random variables and consider sampling-based approaches to estimate their variance. The errors in g, however, result from simplifications or errors made by the designer of the cost model and are out of the scope of this chapter. We show in our experiments that even imperfect cost model functions are useful for estimating uncertainty in predictions.

Based on the idea of treating the c's and the X's as random variables rather than constants, the predicted execution time t_q is then also a random variable so that we can estimate its distribution. A couple of challenges arise immediately. First, unlike the case of providing a point estimate of t_q, knowing that t_q is "some" function of the c's and the X's is insufficient if we want to infer the distribution of t_q — we need to know the *explicit* form of g. By Equation (4.2), g relies on cost functions that map the X's to the n's. As a result, for concreteness we have to choose some specific cost model. Here, for simplicity and generality, we leverage the notion of *logical* cost functions [32] rather than the cost functions of any specific optimizer. The observation is that the costs of an operator can be specified according to its logical execution. For instance, the number of CPU operations of the in-memory *sort* operator could be specified as $n_o = aN \log N$, where N is the input cardinality. Second, while we can show that the distribution of t_q is asymptotically normal based on our current ways of modeling the c's and the X's, determining the parameters of the normal distribution (i.e., the mean and variance) is difficult for non-trivial queries with deep query trees. The challenge arises from correlations between selectivity estimates derived by using shared samples. We present a detailed analysis of the correlations and develop techniques to either directly compute or provide upper bounds for the covariances with respect to the presence of correlations. Finally, providing estimates to distributions of likely running times is desirable only if it can be achieved with low overhead. We show that it is the case for our proposed

techniques — the overhead is almost the same as that of the predictor in Chapter 2 which only provides point estimates.

Since our approach makes a number of approximations when computing the distribution of running time estimates, an important question is how accurate the estimated distribution is. An intuitively appealing experiment is the following: run the query multiple times, measure the distribution of its running times, and see if this matches the estimated distribution. But this is not a reasonable approach due to the subtlety we mentioned earlier. The estimated distribution we calculate is not the expected distribution of the actual query running time, it is the distribution of running times our estimator expects due to uncertainties in its estimation process. To see this another way, note that cardinality estimation error is a major source of running time estimation error. But when the query is actually run, it does not appear at all — the query execution of course observes the true cardinalities, which are identical every time it is run.

Speaking informally, what our predicted running time distribution captures is the "self-awareness" of our estimator. Suppose that embedded in the estimate is a dependence on what our estimator knows is a very inaccurate estimate. Then the estimator knows that while it gives a specific point estimate for the running time (the mean of a distribution), it is likely that the true running time will be far away from the estimate, and it captures this by indicating a distribution with a large variance.

So our task in evaluating our approach is to answer the following question: how closely does the variance of our estimated distribution of running times correspond to the observed errors in our estimates (when compared with true running times)? To answer this question, we estimate the running times for and run a large number of different queries and test the agreement between the observed errors and the predicted distribution of running times, where "agreement" means that larger variations correspond to more inaccurate estimates.

In more detail, we report two metrics over a large number of queries: (M1) the correlation between the standard deviations of the estimated distributions and the actual prediction errors; and (M2) the proximity between the inferred and

observed distributions of prediction errors. We show that (R1) the correlation is strong; and (R2) the two distributions are close. Intuitively, (R1) is *qualitative*; it suggests that one can judge if the prediction errors will be small or large based on the standard deviations of the estimated distributions. (R2) is more *quantitative*; it further suggests that the likelihoods of prediction errors are specified by the distributions as well. We therefore conclude that the estimated distributions do a reasonable job as indicators of prediction errors.

We start by presenting terminology and notation used throughout this chapter in Section 4.2. We then present the details of how to estimate the distributions of the c's and the X's (Section 4.3), the explicit form of g (Section 4.4), and the distribution of t_q (Section 4.5). We further present experimental evaluation results in Section 4.6, discuss related work in Section 4.7, and summarize the chapter in Section 4.8.

4.2 Preliminaries

In most current DBMS implementations, the operators are either unary or binary. Therefore, we can model a query plan with a rooted *binary tree*. Consider an operator O in the plan. We use O_l and O_r to represent its *left* and *right* child operator, and use N_l and N_r to denote its left and right input cardinality. If O is unary, then O_r does not exist and thus $N_r = 0$. We use M to denote O's output cardinality.

Let $\mathcal{T}$ be the subtree rooted at the operator O, and let $\mathcal{R}$ be the (multi)set of relations accessed by the leaf nodes of $\mathcal{T}$. Note that the leaf nodes in a query plan must be *scan* operators that access the underlying tables.[1] We call $\mathcal{R}$ the *leaf tables* of O. Let $|\mathcal{R}| = \prod_{R \in \mathcal{R}} |R|$. We define the *selectivity* X of O to be:

$$X = \frac{M}{|\mathcal{R}|} = \frac{M}{\prod_{R \in \mathcal{R}} |R|}. \tag{4.3}$$

Example 4.1 (Selectivity)**.** *Consider the query plan in Figure 4.1.* O_1*,* O_2*, and* O_3 *are scan operators that access three underlying tables* R_1*,* R_2*, and* R_3*, and* O_4 *and* O_5 *are join*

[1] We use "*relation*" and "*table*" interchangeably in this chapter since our discussion does not depend on the *set*/*bag* semantics.

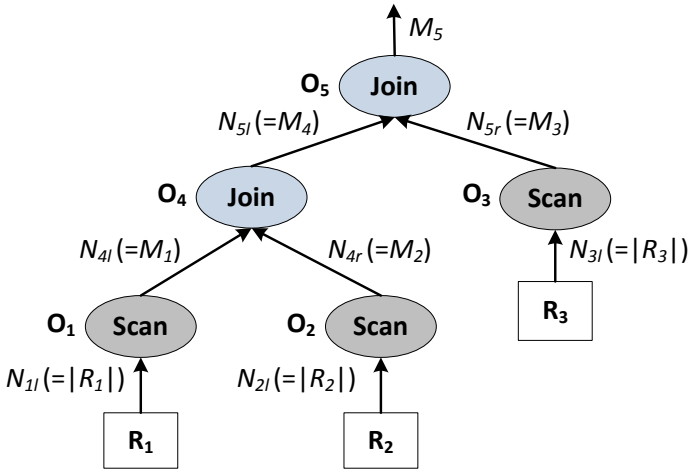


Figure 4.1: Example query plan.

Notation	Description
O	An operator in the query plan
O_l (O_r)	The left (right) child operator of O
N_l (N_r)	The left (right) input cardinality of O
M	The output cardinality of O
$\mathcal{R}$	The leaf tables of O
X	The selectivity of O
$\mathcal{T}$	The subtree rooted at O
$\mathtt{Desc}(O)$	The descendant operators of O in $\mathcal{T}$

Table 4.1: Terminology and notation.

operators. The selectivity of O_1*, for instance, is* $X_1 = \frac{M_1}{|R_1|}$*, whereas the selectivity of* O_4 *is* $X_4 = \frac{M_4}{|R_1|\cdot|R_2|}$.

We summarize the above notation in Table 4.1 for convenience of reference. Since the n's in Equation (4.1) are functions of input/output cardinalities of the operators (we discuss different types of cost functions in Section 4.4.1), it is clear that the n's are also functions of the selectivities (i.e., the X's) defined here. Based on Equation (4.2), t_q is therefore a function of the c's and the X's. We next discuss how to measure the uncertainties in these parameters.

4.3 Input Distributions

To learn the distribution of t_q, we first need to know the distributions of the c's and the X's. We do this by extending the framework in Chapter 2.

4.3.1 Distributions of the c's

In Section 2.3, we designed dedicated calibration queries for each c. As one more example, consider the following:

Example 4.2 (Calibration Query)**.** *Suppose that we want to know the value of* c_t*, namely, the CPU cost of processing one tuple. We can use the calibration query* `SELECT * FROM R`*, where* `R` *is some table whose size is known and is loaded into memory. Since this query only involves* c_t*, its execution time* τ *can be expressed as* $\tau = |R| \cdot c_t$*. We can then run the query, record* τ*, and compute* c_t *from this equation.*

Note that we can use different `R`'s here, and different `R`'s may give us different c_t's. We can think of these observed values as i.i.d. samples from the distribution of c_t, and in Chapter 2 we used the sample mean as our estimate of c_t. To quantify the uncertainty in c_t, it would make more sense to treat c_t as a random variable rather than a constant. We assume that the distribution of c_t is normal (i.e., Gaussian), for intuitively the CPU speed is likely to be stable and centered around its mean value. Now let $c_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$. It is then a common practice to use the mean and variance of the observed c_t's as estimates for μ_t and σ_t^2. In general, we can apply similar arguments to all the five cost units.

4.3.2 Distributions of the X's

The uncertainties in the X's are quite different from those in the c's. The uncertainties in the c's are due to unavoidable fluctuations in hardware execution speeds. In other words, the c's are inherently random. However, the X's are actually fixed numbers — if we run the query we should always obtain the same ground truths for the X's. The uncertainties in the X's really come from the fact that so far we do

not have a perfect selectivity estimator. How to quantify the uncertainties in the X's therefore depends on the nature of the selectivity estimator used. Here we extend the sampling-based approach used in Chapter 2, which was first proposed by Haas et al. [44]. It provides a mathematically rigorous way to quantify potential errors in selectivity estimates. It remains interesting future work to investigate the possibility of extending other alternative estimators such as those based on histograms.

4.3.2.1 A Sampling-Based Selectivity Estimator

For self-containment purpose, in the following we briefly describe the sampling-based selectivity estimator we used. Readers are referred to Section 2.4 for more details. Suppose that we have a database consisting of K relations R_1, ..., R_K, where R_k is partitioned into m_k blocks each with size N_k, namely, $|R_k| = m_k N_k$. Without loss of generality, let q be a selection-join query over R_1, ..., R_K, and let $B(k, j)$ be the j-th block of relation k ($1 \leqslant j \leqslant m_k$, and $1 \leqslant k \leqslant K$). Define

$$\mathbf{B}(L_{1,i_1}, ..., L_{K,i_K}) = B(1, L_{1,i_1}) \times \cdots \times B(K, L_{K,i_K}),$$

where $B(k, L_{k,i_k})$ is the block (with index L_{k,i_k}) randomly picked from the relation R_k in the i_k-th sampling step. After n steps, we can obtain n^K such samples (notice that these samples are not independent), and the estimator is defined as

$$\rho_n = \frac{1}{n^K} \sum_{i_1=1}^{n} \cdots \sum_{i_K=1}^{n} \rho_{\mathbf{B}(L_{1,i_1}, \cdots, L_{K,i_K})}. \tag{4.4}$$

Here ρ_n is the estimated selectivity of q (after n sampling steps), and $\rho_{\mathbf{B}}$ is the observed selectivity of q over the sample $\mathbf{B}$. This estimator is shown to be both unbiased and strongly consistent for the actual selectivity ρ of q [44].[2]

[2]Strong consistency is also called *almost sure convergence* in probability theory (denoted as "a.s."). It means that the more samples we take, the closer ρ_n is to ρ.

By applying the Central Limit Theorem, we can show that

$$\frac{\sqrt{n}}{\sigma}(\rho_n - \rho) \xrightarrow{d} N(0,1).$$

That is, the distribution of ρ_n is approximately normal after a large number of sampling steps [44]: $\rho_n \sim \mathcal{N}(\rho, \sigma_n^2)$, where $\sigma_n^2 = \sigma^2/n$ and $\sigma^2 = \lim_{n\to\infty} n\,\mathrm{Var}[\rho_n]$. We present a more detailed study of $\mathrm{Var}[\rho_n]$ in Appendix A.2.

However, here σ_n^2 is unknown since σ^2 is unknown. In [44], the authors further proposed the following estimator for σ^2:

$$S_n^2 = \sum_{k=1}^{K}\left(\frac{1}{n-1}\sum_{j=1}^{n}(Q_{k,j,n}/n^{K-1} - \rho_n)^2\right), \tag{4.5}$$

for $n \geqslant 2$ (we set $S_1^2 = 0$). Here

$$Q_{k,j,n} = \sum_{(i_1,\ldots,i_K)\in\Omega_k^{(n)}(j)} \rho_{\mathbf{B}(L_{1,i_1},\ldots,L_{K,i_K})}, \tag{4.6}$$

where $\Omega_k^{(n)}(j) = \{(i_1,\ldots,i_K) \in \{1,\ldots,n\}^K : i_k = j\}$. It can be shown that $\lim_{n\to\infty} S_n^2 = \sigma^2$ a.s. As a result, it is reasonable to approximate σ^2 with S_n^2 when n is large. So $\sigma_n^2 \approx S_n^2/n$.

4.3.2.2 Efficient Computation of S_n^2

Efficiency is crucial for a predictor to be practically useful. We have discussed efficient implementation of ρ_n in Section 2.4. Taking samples at runtime might not be acceptable since it will result in too many random disk I/O's. Therefore, we instead take samples off-line and store them as materialized views (i.e., sample tables). In the following presentation, we use R^s to denote the sample table of a relation R. In Section 2.4, we further showed that, given a selection-join query, we can estimate the selectivities of all the selections and joins by running the original query plan over the sample tables once. The trick is that, since the block size is not

specified when partitioning the relations, it could be arbitrary. We can then let a block be a single tuple so that the cross-product of sample blocks is reduced to the cross-product of sample tuples.

Example 4.3 (Implementation of ρ_n)**.** *Let us consider the query plan in Figure 4.1 again. Based on the tuple-level partitioning scheme, by Equation (4.4) we can simply estimate* X_4 *and* X_5 *as*

$$\widehat{X}_4 = \frac{|R_1^s \bowtie R_2^s|}{|R_1^s| \cdot |R_2^s|} \quad \textit{and} \quad \widehat{X}_5 = \frac{|R_1^s \bowtie R_2^s \bowtie R_3^s|}{|R_1^s| \cdot |R_2^s| \cdot |R_3^s|}.$$

Also note that we can compute the two numerators by running the query plan over the sample relations R_1^s, R_2^s, *and* R_3^s *once. That is, to compute* $R_1^s \bowtie R_2^s \bowtie R_3^s$, *we reuse the join results from* $R_1^s \bowtie R_2^s$ *that has been computed when estimating* X_4.

We now extend the above framework to further compute S_n^2. For this sake we need to know how to compute the $Q_{k,j,n}$'s in Equation (4.5). Let us consider the cases when an operator represents a selection (i.e., a scan), a two-way join, or a multi-way join query.

Selection In this case, $K = 1$ and by Equation (4.6) $Q_{k,j,n}$ is reduced to $Q_{1,j,n} = \rho_B(L_{1,j})$. Therefore, S_n^2 can be simplified as

$$S_n^2 = \frac{1}{n-1} \sum_{j=1}^{n} (\rho_B(L_{1,j}) - \rho_n)^2.$$

Since a block here is just a tuple, $\rho_B(L_{1,j}) = 0$ or $\rho_B(L_{1,j}) = 1$. We thus have

$$\begin{aligned} S_n^2 &= \frac{1}{n-1}\Big(\sum_{\rho_B(L_{1,j})=0} \rho_n^2 + \sum_{\rho_B(L_{1,j})=1} (1-\rho_n)^2 \Big) \\ &= \frac{1}{n-1}\big((n-M)\rho_n^2 + M(1-\rho_n)^2\big), \end{aligned}$$

where M is the number of output tuples from the selection. When n is large, $n \approx n-1$, so we have

$$S_n^2 \approx (1-\frac{M}{n})\rho_n^2 + \frac{M}{n}(1-\rho_n)^2 = \rho_n(1-\rho_n),$$

by noticing that $\rho_n = \frac{M}{n}$. Hence S_n^2 is directly computable for a scan operator once we know its estimated selectivity ρ_n.

Two-way Join Consider a join $R_1 \bowtie R_2$. In this case, $Q_{k,j,n}$ $(k = 1,2)$ can be reduced to

$$Q_{1,j,n} = \sum_{i_2=1}^{n} \rho_{\mathbf{B}}(L_{1,j}, L_{2,i_2}) \text{ and } Q_{2,j,n} = \sum_{i_1=1}^{n} \rho_{\mathbf{B}}(L_{1,i_1}, L_{2,j}).$$

Again, since a block here is just a tuple, $\rho_{\mathbf{B}}$ is either 0 or 1. It is then equivalent to computing the following two quantities:

- $Q_{1,j,n} = |\{t_{1j}\} \bowtie R_2^s|$, where t_{1j} is the jth tuple of R_1^s;
- $Q_{2,j,n} = |R_1^s \bowtie \{t_{2j}\}|$, where t_{2j} is the jth tuple of R_2^s.

That is, to compute $Q_{k,j,n}$ $(k = 1,2)$, conceptually we need to join each sample tuple of one relation with all the sample tuples of the other relation. However, directly performing this is quite expensive, for we need to do $2n$ joins here.

We seek a more efficient solution. Recall that we need to join R_1^s and R_2^s to compute ρ_n. Let $R^s = R_1^s \bowtie R_2^s$. Consider any $t \in R^s$. t must satisfy $t = t_{1i} \bowtie t_{2j}$, where $t_{1i} \in R_1^s$ and $t_{2j} \in R_2^s$. Then t contributes 1 to $Q_{1,i,n}$ and 1 to $Q_{2,j,n}$. On the other hand, any t in $R_1^s \times R_2^s$ but not in R^s will contribute nothing to the Q's. Based on this observation, we only need to scan the tuples in R^s and increment the corresponding Q's. The remaining problem is how to know the indexes i and j as in $t = t_{1i} \bowtie t_{2j}$. For this purpose, we assign an *identifier* to each tuple in the sample tables when taking the samples. This is akin to the idea in data provenance research where tuples are annotated to help tracking the lineages of the query results [41].

Multi-way Joins The approach of processing two-way joins can be easily generalized to handle multi-way joins. Now we have

$$Q_{k,j,n} = |R_1^s \bowtie \cdots \bowtie \{t_{kj}\} \bowtie \cdots \bowtie R_K^s|.$$

As a result, if we let $R^s = R_1^s \bowtie \cdots \bowtie R_K^s$, then any $t \in R^s$ satisfies $t = t_{1i_1} \bowtie \cdots \bowtie t_{Ki_K}$. $t \in R_1^s \times \cdots \times R_K^s$ will contribute 1 to each $Q_{k,i_k,n}$ ($1 \leqslant k \leqslant K$) if and only if $t \in R^s$. Therefore, as before, we can just simply scan R^s and increment the corresponding Q's when processing each tuple.

Putting It Together Algorithm 4 summarizes the procedure of computing ρ_n and S_n^2 for a single operator O. It is straightforward to incorporate it into the previous framework where the selectivities of the operators are refined in a bottom-up fashion (recall Algorithm 1). We discuss some implementation details in the following.

First, the selectivity estimator cannot work for operators such as *aggregates*. Our current strategy is to use the original cardinality estimates from the optimizer to compute ρ_n, and we simply set S_n^2 to be 0 for these operators (lines 3 to 5). This may cause inaccuracy in the prediction as well as our estimate of its uncertainty, if the optimizer does a poor job in estimating the cardinalities. However, we find that it works reasonably well in our experiments. Nonetheless, it is interesting future work to incorporate sampling-based estimators for aggregates (e.g., the GEE estimator [19]) into our current framework.

Second, to compute the $Q_{k,i_k,n}$'s, we maintain a hash map H_k for each k with i_k's the keys and $Q_{k,i_k,n}$'s the values. The size of H_k is upper bounded by $|R_k^s|$ and usually is much smaller.

Third, for simplicity of exposition, in Algorithm 4 we first compute the whole R^s and then scan it. In practice we actually do not need to do this. Typical join operators, such as *merge join*, *hash join*, and *nested-loop join*, usually compute join results on the fly. Once a join tuple is computed, we can immediately postprocess it by increasing the corresponding $Q_{k,i_k,n}$'s. Therefore, we can avoid the additional

Algorithm 4: Computation of ρ_n and S_n^2.

Input: O, an operator; $\mathcal{R}^s = \{R_1^s, ..., R_K^s\}$, the sample tables; Agg, if some $O' \in Desc(O)$ is an aggregate
Output: ρ_n, estimated selectivity of O; S_n^2, sample variance

$R^s \leftarrow RunOperator(O, \mathcal{R}^s)$;
if Agg **then**
  $M \leftarrow CardinalityByOptimizer(O)$;
  $\rho_n \leftarrow \frac{M}{\prod_{k=1}^{K} |R_k|}$;
  $S_n^2 \leftarrow 0$;
else if O *is a scan* **then**
  $\rho_n \leftarrow \frac{|R^s|}{|R_1^s|}$;
  $S_n^2 \leftarrow \rho_n(1-\rho_n)$;
else if O *is a join* **then**
  $\rho_n \leftarrow \frac{|R^s|}{\prod_{k=1}^{K} |R_k^s|}$;
  foreach $t = t_{1i_1} \bowtie \cdots \bowtie t_{Ki_K} \in R^s$ **do**
    $Q_{k,i_k,n} \leftarrow Q_{k,i_k,n} + 1$, for $1 \leqslant k \leqslant K$;
  end
  $S_n^2 \leftarrow \sum_{k=1}^{K} \left(\frac{1}{n-1} \sum_{j=1}^{n} (Q_{k,j,n}/n^{K-1} - \rho_n)^2 \right)$;
else
  $\rho_n \leftarrow \hat{\mu}_l$, $S_n^2 \leftarrow \hat{\sigma}_l^2$; // Let $X_l \sim \mathcal{N}(\hat{\mu}_l, \hat{\sigma}_l^2)$.
end
return ρ_n and S_n^2;

memory overhead of caching intermediate join results, which might be large even if the sample tables are small.

4.4 Cost Functions

By Equation (4.2), to infer the distribution of t_q for a query q, we also need to know the explicit form of g. According to Equation (4.1), g relies on the cost functions of operators that map the selectivities to the n's. As mentioned in Section 4.1, we use logical cost functions in our work. While different DBMS may differ in their

implementations of a particular operator, e.g., nested-loop join, they follow the same execution logic and therefore have the same logical cost function. In the following, we first present a detailed study of representative cost functions. We then formulate the computation of cost functions as an optimization problem that seeks the best fit for the unknown coefficients, and we use standard quadratic programming techniques to solve this problem.

4.4.1 Types of Functions

We consider the following types of cost functions in this chapter:

(C1) $f = a_0$: The cost function is a constant. For instance, since a sequential scan has no random disk reads, $n_r = 0$.

(C2) $f = a_0 M + a_1$: The cost function is linear with respect to the *output* cardinality. For example, the number of random reads of an index-based table scan falls into this category, which is proportional to the number of qualified tuples that pass the selection predicate.

(C3) $f = a_0 N_l + a_1$: The cost function is linear with respect to the *input* cardinality. This happens for unary operators that process each input tuple once. For example, *materialization* is such an operator that creates a buffer to cache the intermediate results.

(C4) $f = a_0 N_l^2 + a_1 N_l + a_2$: The cost function is nonlinear with respect to the *input* cardinality. For instance, the number of CPU operations (i.e., c_o) performed by a *sort* operator is proportional to $N_l \log N_l$. While different nonlinear unary operators may have specific cost functions, we choose to only use quadratic polynomials based on the following observations:

- It is quite general to approximate the nonlinear cost functions used by current relational operators. First, as long as a function is smooth (i.e., it has continuous derivatives up to some desired order), it can be approximated by using the well-known Taylor series, which is basically

a polynomial of the input variable. Second, for efficiency reasons, the overhead of an operator usually does not go beyond quadratic of its input cardinality — we are not aware of any operator implementation whose time complexity is $\omega(N^2)$. Similar observations have been made in [30].

- Compared with functions such as logarithmic ones, polynomials are mathematically much easier to manipulate. Since we need to further infer the distribution of the predicted query execution time based on the cost functions, this greatly simplifies the derivations.

(C5) $f = a_0 N_l + a_1 N_r + a_2$: This cost function is linear with respect to the *input* cardinalities when the operator is binary. An interesting observation here is that the cost functions in the case of binary operators are not necessarily nonlinear. For example, the number of I/O's involved in a hash join is only proportional to the number of input tuples.

(C6) $f = a_0 N_l N_r + a_1 N_l + a_2 N_r + a_3$: The cost function here also involves the product of the left and right input cardinalities of a binary operator. This happens typically in a nested-loop join, which iterates over the inner (i.e., the right) input table multiple times with respect to the number of rows in the outer (i.e., the left) input table.

It is straightforward to translate these cost functions in terms of selectivities. Specifically, we have $N_l = |\mathcal{R}_l|X_l$, $N_r = |\mathcal{R}_r|X_r$, and $M = |\mathcal{R}|X$. The above six cost functions can be rewritten as:

(C1′) $f = b_0$, where $b_0 = a_0$.

(C2′) $f = b_0 X + b_1$, where $b_0 = a_0|\mathcal{R}|$ and $b_1 = a_1$.

(C3′) $f = b_0 X_l + b_1$, where $b_0 = a_0|\mathcal{R}_l|$ and $b_1 = a_1$.

(C4′) $f = b_0 X_l^2 + b_1 X_l + b_2$, where $b_0 = a_0|\mathcal{R}_l|^2$, $b_1 = a_1|\mathcal{R}_l|$, and $b_2 = a_2$.

(C5′) $f = b_0 X_l + b_1 X_r + b_2$, where $b_0 = a_0|\mathcal{R}_l|$, $b_1 = a_1|\mathcal{R}_r|$, and $b_2 = a_2$.

(C6′) $f = b_0 X_l X_r + b_1 X_l + b_2 X_r + b_3$, where $b_0 = a_0|\mathcal{R}_l| \cdot |\mathcal{R}_r|$, $b_1 = a_1|\mathcal{R}_l|$, $b_2 = a_2|\mathcal{R}_r|$, and $b_3 = a_3$.

4.4.2 Computation of Cost Functions

To compute the cost functions, we use an approach that is similar to the one proposed in [30]. Regarding the types of cost functions we considered, the only unknowns given the selectivity estimates are the coefficients in the functions (i.e., the b's). Moreover, notice that f is a *linear* function of the b's once the selectivities are given. We can then collect a number of f values by feeding in the cost model with different X's and find the best fit for the b's.

As an example, consider (C4′). Suppose that we invoke the cost model m times and obtain m points:

$$\{(X_{l1}, f_1), ..., (X_{lm}, f_m)\}.$$

Let $\mathbf{y} = (f_1, ..., f_m)$, $\mathbf{b} = (b_0, b_1, b_2)$, and

$$\mathbf{A} = \begin{pmatrix} X_{l1}^2 & X_{l1} & 1 \\ \vdots & \vdots & \vdots \\ X_{lm}^2 & X_{lm} & 1 \end{pmatrix}.$$

The optimization problem we are concerned with is:

$$\begin{aligned} \underset{\mathbf{b}}{\text{minimize}} \quad & \|\mathbf{Ab} - \mathbf{y}\| \\ \text{subject to} \quad & b_i \geqslant 0,\ i = 0, 1. \end{aligned}$$

Note that we require b_0 and b_1 be nonnegative since they have the natural semantics in the cost functions as the amount of work with respect to the corresponding terms. For example, $b_1 X_l = a_1 N_l$ is the amount of work that is proportional to the input cardinality. To solve this quadratic programming problem, we use the `qpsolve` function of Scilab [81]. Other equivalent solvers could also be used.

The remaining problem is how to pick these (X, f)'s. In theory, one could arbitrarily pick the X's from $[0, 1]$ to obtain the corresponding f's as long as we have

more points than unknowns. Although more points usually mean we can have better fittings, in practice we cannot afford too many points due to the efficiency requirements when making the prediction. On the other hand, given that the X's here follow normal distributions and the variances are usually small when the sample size is large, the likely selectivity estimates are usually concentrated in a much shorter interval than $[0, 1]$. Intuitively, we should take more points within this interval, for we can then have a more accurate view of the shape of the cost function restricted to this interval. Therefore, in our current implementation, we adopt the following strategy.

Let $X \sim \mathcal{N}(\mu, \sigma^2)$. Consider the interval $\mathcal{I} = [\mu - 3\sigma, \mu + 3\sigma]$. It is well known that $\Pr(X \in \mathcal{I}) \approx 0.997$, which means the probability that X falls out of $\mathcal{I}$ is less than 0.3%. We then proceed by partitioning $\mathcal{I}$ into W subintervals of equal width, and pick the $W + 1$ boundary X's to invoke the cost model. Generalizing this idea to binary cost functions is straightforward.

Suppose $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$ and $X_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$. Let $\mathcal{I}_l = [\mu_l - 3\sigma_l, \mu_l + 3\sigma_l]$ and $\mathcal{I}_r = [\mu_r - 3\sigma_r, \mu_r + 3\sigma_r]$. We then partition $\mathcal{I}_l \times \mathcal{I}_r$ into a $W \times W$ grid and obtain $(W + 1) \times (W + 1)$ points (X_l, X_r) to invoke the cost model.

4.5 Distribution of Running Times

We have discussed how to estimate the distributions of input parameters (i.e., the c's and the X's) and how to estimate the cost functions of each operator. In this section, we discuss how to combine these two to further infer the distribution of t_q for a query q.

Since $t_q = g(\mathbf{c}, \mathbf{X})$, the distribution of t_q relies on the *joint* distribution of $(\mathbf{c}, \mathbf{X})$.[3] We therefore first present a detailed analysis of the correlations between the c's and the X's. Based on that, we then show that the distribution of t_q is asymptotically normal and thus reduce the problem to estimating the two parameters of normal

[3]Note that the distributions of the c's and X's that we obtained in Section 4.3 are *marginal* rather than joint distributions.

distributions, i.e., the mean and variance of t_q. We further address the nontrivial problem of computing $\mathrm{Var}[t_q]$ due to correlations between selectivity estimates.

4.5.1 Correlations of Input Variables

In our current setting, it is reasonable to assume that the c's and the X's are independent. We next analyze the correlations within the c's and the X's.

4.5.1.1 Correlations Between Cost Units

Since the randomness within the c's comes from the variations in hardware execution speeds, by using our current framework we have no way to observe the true values of the c's and thus it is impossible to obtain the exact joint distribution of the c's. Nonetheless, it might be reasonable to assume the independence of the c's. First, since the CPU and I/O cost units measure the speeds of different hardware devices, their values do not depend on each other. Second, within each group (i.e., CPU or I/O cost units), we used independent calibration queries for each individual cost unit.

Assumption 1. *The c's are independent of each other.*

We further note here that the independence of the c's depends on the cost model as well as the hardware configurations. For instance, if certain devices are connected via the same infrastructure (e.g., a bus), then they might influence each other's communication patterns. Our current framework for calibrating the c's cannot capture the correlations of the c's. However, perhaps low-level tools for monitoring hardware execution status could be used for this purpose. We leave it as interesting future work to investigate such possibilities and study the effectiveness of incorporating correlation information of the c's into our current framework.

4.5.1.2 Correlations Between Selectivity Estimates

The X's are clearly not independent, because the same samples are used to estimate the selectivities of different operators. In the following, we study the correlations between the X's in detail.

Let O and O′ be two operators, and $\mathcal{R}$ and $\mathcal{R}'$ be the corresponding leaf tables. Consider the two corresponding selectivity estimates ρ_n and ρ'_n as defined by Equation (4.4). Since the samples from each table are drawn independently, we first have the following observation:

Lemma 4.4. *If* $\mathcal{R} \cap \mathcal{R}' = \emptyset$, *then* $\rho_n \perp \rho'_n$.[4]

For binary operators, the next result follows from Lemma 4.4 immediately:

Lemma 4.5. *Let* O *be binary. If* $\mathcal{R}_l \cap \mathcal{R}_r = \emptyset$, *then* $X_l \perp X_r$.

That is, X_l and X_r will only be correlated if $\mathcal{R}_l$ and $\mathcal{R}_r$ share *common* relations. However, in practice, we can maintain more than one sample table for each relation. When the database is large, this is affordable since the number of samples is very small compared to the database size (see Section 2.5). Since the samples from each relation are drawn independently, X_l and X_r are still independent if we use a different sample table for each appearance of a shared relation. We thus assume $X_l \perp X_r$ in the rest of the chapter.

More generally, X and X′ are independent as long as neither $O \in Desc(O')$ nor $O' \in Desc(O)$. However, the above discussion cannot be applied if $O \in Desc(O')$ (or vice versa). This is because we pass the join results from downstream joins to upstream joins when estimating the selectivities (recall Example 4.3). So $\mathcal{R}$ and $\mathcal{R}'$ are naturally not disjoint. In fact, $\mathcal{R} \subseteq \mathcal{R}'$. To make ρ_n and ρ'_n independent, we need to replace each of the sample tables used in computing ρ'_n with another sample table from the same relation, which basically is the same as run the query plan again on a different set of sample tables. The number of runs is then in proportion to the number of selective operators (i.e., selections and joins) in the query plan,

[4] We use $Y \perp Z$ to denote that Y and Z are independent.

and the runtime overhead might be prohibitive in practice. We summarize this observation as follows:

Lemma 4.6. *Given that multiple sample tables of the same relation can be used,* ρ_n *and* ρ'_n *are correlated if and only if either* $O \in \mathtt{Desc}(O')$ *or vice versa.*

4.5.2 Asymptotic Distributions

Now for specificity suppose that the query plan of q contains m operators O_1, ..., O_m. Since t_q is the sum of the predicted execution time spent on each operator, it can be expressed as $t_q = \sum_{k=1}^{m} t_k$, where t_k is the predicted execution time of O_k and is itself a random variable.

We next show that t_k is asymptotically normal, and then by using very similar arguments, we can show that t_q is asymptotically normal as well. Since t_k can be further expressed in terms of Equation (4.1), to learn its distribution we need to know the distributions of cost functions that map the selectivities to the n's. We therefore start by discussing the distributions of the typical cost functions as presented in Section 4.4.1.

4.5.2.1 Asymptotic Distributions of Cost Functions

In the following discussion, we assume that $X \sim \mathcal{N}(\mu, \sigma^2)$, $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$, and $X_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$. The distributions of the six types of cost functions previously discussed are as follows:

(C1′) $f = b_0$: $f \sim \mathcal{N}(b_0, 0)$.

(C2′) $f = b_0 X + b_1$: $f \sim \mathcal{N}(b_0\mu + b_1, b_0^2\sigma^2)$.

(C3′) $f = b_0 X_l + b_1$: $f \sim \mathcal{N}(b_0\mu_l + b_1, b_0^2\sigma_l^2)$.

(C4′) $f = b_0 X_l^2 + b_1 X_l + b_2$: In this case $\Pr(f)$ is not normal. Although it is possible to derive the *exact* distribution of f based on the distribution of X_l, the derivation would be very messy. Instead, we consider $f^{\mathcal{N}} \sim \mathcal{N}(\mathrm{E}[f], \mathrm{Var}[f])$ and use

k	Non-central moment $E(X^k)$
1	μ
2	$\mu^2 + \sigma^2$
3	$\mu^3 + 3\mu\sigma^2$
4	$\mu^4 + 6\mu^2\sigma^2 + 3\sigma^4$

Table 4.2: Non-central moments of $X \sim N(\mu, \sigma^2)$.

this to approximate $\Pr(f)$. We present the formula of $\mathrm{Var}[f]$ in Lemma 4.7 below. Obviously, $f^{\mathcal{N}}$ and f have the same expected value and variance. Moreover, we can actually show that $f^{\mathcal{N}}$ and f (and therefore their corresponding distributions) are very close to each other when the number of samples is large (see Theorem 4.8 below).

(C5′) $f = b_0 X_l + b_1 X_r + b_2$: Since $X_l \perp X_r$ by Lemma 4.5, $f \sim \mathcal{N}(b_0\mu_l + b_1\mu_r + b_2, b_0^2\sigma_l^2 + b_1^2\sigma_r^2)$.

(C6′) $f = b_0 X_l X_r + b_1 X_l + b_2 X_r + b_3$: Again, $\Pr(f)$ is not normal. Since $X_l \perp X_r$, $X_l X_r$ follows the so called *normal product distribution* [13], whose exact form is again complicated. We thus use the same strategy as in (C4′) (see Lemma 4.10 and Theorem 4.11 below).

Lemma 4.7. *If* $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$ *and* $f = b_0 X_l^2 + b_1 X_l + b_2$, *then*

$$\mathrm{Var}[f] = \sigma_l^2[(b_1 + 2b_0\mu_l)^2 + 2b_0^2\sigma_l^2].$$

Proof. Table 4.2 presents the non-central moments of a normal variable $X \sim \mathcal{N}(\mu, \sigma^2)$. By Table 4.2, $\mathrm{Var}[X_l^2] = 2\sigma_l^2(2\mu_l^2 + \sigma_l^2)$, and $\mathrm{Cov}(X_l^2, X_l) = 2\mu_l\sigma_l^2$. Thus

$$\begin{aligned} \mathrm{Var}[f] &= b_0^2 \mathrm{Var}[X_l^4] + b_1^2 \mathrm{Var}[X_l] + 2b_0 b_1 \mathrm{Cov}(X_l^2, X_l) \\ &= \sigma_l^2[(b_1 + 2b_0\mu_l)^2 + 2b_0^2\sigma_l^2]. \end{aligned}$$

This completes the proof of the lemma. □

Theorem 4.8. *Suppose that* $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$ *and* $f = b_0 X_l^2 + b_1 X_l + b_2$. *Let* $f^{\mathcal{N}} \sim \mathcal{N}(E[f], \mathrm{Var}[f])$, *where* $\mathrm{Var}[f]$ *is shown in Lemma 4.7. Then* $f^{\mathcal{N}} \xrightarrow{p} f$.[5]

To prove the theorem, we need the following result:

Theorem 4.9. $\mathrm{Var}[\rho_n]$ *in Equation (A.1) can be bounded as:*

$$\mathrm{Var}[\rho_n] \leqslant \big(1 - (1 - \frac{1}{n})^K\big)\rho(1-\rho).$$

Proof. The proof is straightforward since this is a special case of Theorem A.10 (see Appendix A.4). Specifically, we have $\mathrm{Var}[\rho_n] = \mathrm{Cov}(\rho_n, \rho_n)$. By letting $m = K$ in Theorem A.10, we obtain $\mathrm{Var}[\rho_n] \leqslant f(n, K) g(\rho)^2$, where $f(n, K) = 1 - (1 - \frac{1}{n})^K$ and $g(\rho) = \sqrt{\rho(1-\rho)}$. This completes the proof. □

By Theorem 4.9, $\mathrm{Var}[\rho_n] \to 0$ as $n \to \infty$. We are now ready to prove Theorem 4.8:

Proof. (of Theorem 4.8) Let $\mu_l = \rho_n$, and $E[\rho_n] = \rho$. Define $g(X) = b_0 X^2 + b_1 X + b_2$. Since ρ_n is strongly consistent, $\rho_n \xrightarrow{as} \rho$. Moreover, since g is continuous, $f = g(\rho_n) \xrightarrow{as} g(\rho)$ by the continuous mapping theorem. Note that $g(\rho)$ is a constant. On the other hand, by Lemma 4.7 and Theorem 4.9, $\mathrm{Var}[f] \to 0$ as $n \to \infty$. Hence,

$$f^{\mathcal{N}} \xrightarrow{d} E[f] = g(\rho).$$

Since $f \xrightarrow{as} g(\rho)$ implies $f \xrightarrow{p} g(\rho)$,

$$f^{\mathcal{N}} - f \xrightarrow{d} g(\rho) - g(\rho) = 0$$

by Slutsky's theorem. Since 0 is a constant, $f^{\mathcal{N}} - f \xrightarrow{p} 0$ as well. As a result, we have $f^{\mathcal{N}} \xrightarrow{p} f$. □

Lemma 4.10. *If* $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$, $X_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$, *and* $f = b_0 X_l X_r + b_1 X_l + b_2 X_r + b_3$, *then we have*

$$\mathrm{Var}[f] = \sigma_l^2\big(b_0\mu_r + b_1\big)^2 + \sigma_r^2\big(b_0\mu_l + b_2\big)^2 + b_0^2\sigma_l^2\sigma_r^2.$$

[5] $f^{\mathcal{N}} \xrightarrow{p} f$ means $f^{\mathcal{N}}$ converges in probability to f.

Proof. Since $X_l \perp X_r$, $\mathrm{Cov}(X_l, X_r) = 0$. So

$$\begin{aligned}\mathrm{Var}[f] &= b_0^2 \cdot \mathrm{Var}[X_l X_r] + b_1^2\sigma_l^2 + b_2^2\sigma_r^2 \\ &+ 2b_0b_1 \cdot \mathrm{Cov}(X_l X_r, X_l) \\ &+ 2b_0b_2 \cdot \mathrm{Cov}(X_l X_r, X_r).\end{aligned}$$

Since

$$\mathrm{Var}[X_l X_r] = \mu_l^2\sigma_r^2 + \mu_r^2\sigma_l^2 + \sigma_l^2\sigma_r^2,$$

$$\mathrm{Cov}(X_l X_r, X_l) = \mu_r\sigma_l^2,$$

and similarly,

$$\mathrm{Cov}(X_l X_r, X_r) = \mu_l\sigma_r^2,$$

we can have the desired expression for $\mathrm{Var}[f]$ by substituting these quantities. □

Theorem 4.11. *Suppose that* $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$, $X_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$, *and* $f = b_0X_lX_r + b_1X_l + b_2X_r + b_3$. *Let* $f^{\mathcal{N}} \sim \mathcal{N}(\mathrm{E}[f], \mathrm{Var}[f])$, *where* $\mathrm{Var}[f]$ *is shown in Lemma 4.10. Then* $f^{\mathcal{N}} \xrightarrow{p} f$.

Proof. Let $\mu_l = \rho_n$ and $\mu_r = \rho'_n$. Suppose that $\mathrm{E}[\rho_n] = \rho$ and $\mathrm{E}[\rho'_n] = \rho'$. Define

$$g(X, Y) = b_0XY + b_1X + b_2Y + b_3.$$

Since μ_l and μ_r are both strongly consistent, $\rho_n \xrightarrow{as} \rho$ and $\rho'_n \xrightarrow{as} \rho'$. Moreover, since g is continuous, by the continuous mapping theorem we have

$$f = g(\rho_n, \rho'_n) \xrightarrow{as} g(\rho, \rho').$$

Note that $g(\rho, \rho')$ is a constant. On the other hand, by Lemma 4.10 and Theorem 4.9, $\mathrm{Var}[f] \to 0$ as $n \to \infty$. As a result, since $X_l \perp X_r$ by Lemma 4.5, it follows that

$$f^{\mathcal{N}} \xrightarrow{d} \mathrm{E}[f] = g(\rho, \rho').$$

Since $f \xrightarrow{as} g(\rho, \rho')$ implies $f \xrightarrow{p} g(\rho, \rho')$,

$$f^{\mathcal{N}} - f \xrightarrow{d} g(\rho, \rho') - g(\rho, \rho') = 0$$

by Slutsky's theorem. Since 0 is a constant, $f^{\mathcal{N}} - f \xrightarrow{p} 0$ as well. As a result, we have $f^{\mathcal{N}} \xrightarrow{p} f$. □

4.5.2.2 Asymptotic Distribution of t_k

Based on the previous analysis, the cost functions (or equivalently, the n's in Equation (4.1)) are asymptotically normal. Since the c's are normal and independent of the X's (and hence the n's as well), by Equation (4.1) again t_k is asymptotically the sum of products of two independent normal random variables. Specifically, let $\mathcal{C} = \{c_s, c_r, c_t, c_i, c_o\}$, and for $c \in \mathcal{C}$, let f_{kc} be the cost function indexed by c. Defining $t_{kc} = f_{kc}^{\mathcal{N}} c$, we have

$$t_k \approx \sum_{c \in \mathcal{C}} t_{kc} = \sum_{c \in \mathcal{C}} f_{kc}^{\mathcal{N}} c.$$

Again, each t_{kc} is not normal. But we can apply techniques similar to that in Theorem 4.8 here by using the normal random variable

$$t_{kc}^{\mathcal{N}} \sim \mathcal{N}(E[f_{kc}^{\mathcal{N}} c], \mathrm{Var}[f_{kc}^{\mathcal{N}} c]) = \mathcal{N}(E[f_{kc} c], \mathrm{Var}[f_{kc} c])$$

as an approximation of t_{kc}. Defining $Z = E[f_{kc}]c$, we have

Theorem 4.12. $t_{kc} \xrightarrow{d} Z$, *and* $t_{kc}^{\mathcal{N}} \xrightarrow{d} Z$.

Proof. Since f_{kc} and c are independent, we have

$$E[t_{kc}] = E[f_{kc}^{\mathcal{N}} c] = E[f_{kc}]\, E[c]$$

and

$$\mathrm{Var}[t_{kc}] = E^2[f_{kc}]\, \mathrm{Var}[c] + E^2[c]\, \mathrm{Var}[f_{kc}] + \mathrm{Var}[c]\, \mathrm{Var}[f_{kc}].$$

Since $\text{Var}[f_{kc}] \to 0$ as $n \to \infty$,

$$\Pr(t_{kc}^{\mathcal{N}}) \to \mathcal{N}(E[f_{kc}]\,E[c], E^2[f_{kc}]\,\text{Var}[c]).$$

In other words, $t_{kc}^{\mathcal{N}} \xrightarrow{d} E[f_{kc}]c$.

On the other hand, $f_{kc}^{\mathcal{N}} \xrightarrow{p} E[f_{kc}]$ and $c \xrightarrow{p} c$. As a result, we have

$$(f_{kc}^{\mathcal{N}}, c) \xrightarrow{p} (E[f_{kc}], c).$$

By the continuous mapping theorem, $f_{kc}^{\mathcal{N}} c \xrightarrow{p} E[f_{kc}]c$. That is, $t_{kc} \xrightarrow{p} E[f_{kc}]c$, which implies $t_{kc} \xrightarrow{d} E[f_{kc}]c$. This completes the proof of the theorem. □

Theorem 4.12 implies that t_{kc} and $t_{kc}^{\mathcal{N}}$ tend to follow the same distribution as the sample size grows. Since c is normal, Z is normal as well. Furthermore, the independence of the c's also implies the independence of the Z's. So t_k is approximately the sum of the independent normal random variables $t_{kc}^{\mathcal{N}}$. Hence t_k is itself approximately normal with large sample size.

4.5.2.3 Asymptotic Distribution of t_q

Finally, let us consider the distribution of t_q. Since t_q is merely the sum of the t_k's, we have exactly the same situation as when we analyze each t_k. Specifically, we can express t_q as

$$t_q = \sum_{k=1}^{m} t_k \approx \sum_{c \in \mathcal{C}} g_c c,$$

where $g_c = \sum_{k=1}^{m} f_{kc}^{\mathcal{N}}$ is the sum of the cost functions of the operators with respect to the particular c. However, since the $f_{kc}^{\mathcal{N}}$'s are not independent, g_c is not normal. We can again use the normal random variable

$$g_c^{\mathcal{N}} \sim \mathcal{N}(E[g_c], \text{Var}[g_c])$$

as an approximation of g_c. We show $g_c^{\mathcal{N}} \xrightarrow{p} g_c$ in Theorem 4.13 below. With exactly the same argument used in Section 4.5.2.2 we can then see that t_q is approximately normal when the sample size is large.

Theorem 4.13. *Let* $g_c = \sum_{k=1}^{m} f_{kc}^{\mathcal{N}}$ *and*

$$g_c^{\mathcal{N}} \sim \mathcal{N}(E[g_c], \mathrm{Var}[g_c]).$$

Then $g_c^{\mathcal{N}} \xrightarrow{p} g_c$.

Proof. Since by definition $f_{kc}^{\mathcal{N}} \sim \mathcal{N}(E[f_{kc}], \mathrm{Var}[f_{kc}])$ and $\mathrm{Var}[f_{kc}] \to 0$, $f_{kc}^{\mathcal{N}} \xrightarrow{d} E[f_{kc}]$. Since $E[f_{kc}]$ is a constant, it implies that $f_{kc}^{\mathcal{N}} \xrightarrow{p} E[f_{kc}]$. By the continuous mapping theorem, $g_c \xrightarrow{p} \sum_{k=1}^{m} E[f_{kc}]$. On the other hand, since $\mathrm{Var}[g_c] \to 0$, $g_c^{\mathcal{N}} \xrightarrow{d} E[g_c]$. Since $E[g_c]$ is again a constant, it follows that

$$g_c^{\mathcal{N}} \xrightarrow{p} E[g_c] = \sum_{k=1}^{m} E[f_{kc}].$$

As a result, by applying the continuous mapping theorem again, we have $g_c^{\mathcal{N}} - g_c \xrightarrow{p} 0$ and hence $g_c^{\mathcal{N}} \xrightarrow{p} g_c$. □

4.5.2.4 Discussion

The analysis that t_q is asymptotically normal relies on three facts: (1) the selectivity estimates are unbiased and strongly consistent; (2) the cost model is additive; and (3) the cost units are independently normally distributed. While the first fact is a property of the sampling-based selectivity estimator and thus always holds, the latter two are specific merits of the cost model of PostgreSQL, though we believe that cost models of other database systems share more or less similar features. (As far as we know, MySQL [97], IBM DB2 [1], Oracle [2], and Microsoft SQL Server [3] use similar cost models.) Therefore, we need new techniques when either (2) or (3) does not hold. For instance, if the cost model is still additive and the c's are independent but cannot be modeled as normal variables, then by the analysis in

Section 4.5.2.3 we can still see that t_q is asymptotically a linear combination of the c's and thus the distribution of t_q can be expressed in terms of the *convolution* of the distributions of the c's. We may then find this distribution by using generating functions or characteristic functions [79]. We leave the investigation of other types of cost models as future work.

4.5.3 Computing Distribution Parameters

As discussed, we can approximate the distribution of t_q with a normal distribution $\mathcal{N}(E[t_q], \mathrm{Var}[t_q])$. We are then left with the problem of estimating the two parameters $E[t_q]$ and $\mathrm{Var}[t_q]$. While $E[t_q]$ is trivial to compute — it is merely the original prediction from our predictor, estimating $\mathrm{Var}[t_q]$ is a challenging problem due to the correlations presented in selectivity estimates.

In more detail, so far we have observed the additive nature of t_q, that is, $t_q = \sum_{k=1}^{m} t_k$ and $t_k = \sum_{c \in \mathcal{C}} t_{kc}$ (Section 4.5.2.2). Recall the fact that for sum of random variables $Y = \sum_{1 \leqslant i \leqslant m} Y_i$,

$$\mathrm{Var}[Y] = \sum\nolimits_{1 \leqslant i,j \leqslant m} \mathrm{Cov}(Y_i, Y_j).$$

Applying this to t_q, our task is then to compute each $\mathrm{Cov}(t_i, t_j)$. Note that

$$\mathrm{Cov}(t_i, t_i) = \mathrm{Var}[t_i],$$

which is easy to compute. So it is left to compute $\mathrm{Cov}(t_i, t_j)$ for $i \neq j$. By linearity of covariance,

$$\mathrm{Cov}(t_i, t_j) = \mathrm{Cov}\left(\sum_{c \in \mathcal{C}} t_{ic}, \sum_{c \in \mathcal{C}} t_{jc}\right) = \sum_{c,c' \in \mathcal{C}} \mathrm{Cov}(t_{ic}, t_{jc'}).$$

In the following discussion, we first specify the cases where direct computation of $\mathrm{Cov}(t_{ic}, t_{jc'})$ can be done. We then develop upper bounds for those covariances that cannot be directly computed.

4.5.3.1 Direct Computation of Covariances

Any $\mathrm{Cov}(t_{ic}, t_{jc'})$ can fall into the following two cases:

- $i = j$, the covariance between different cost functions from the same operator;
- $i \neq j$, the covariance between cost functions from different operators.

Consider the case $i = j$ first. If the operator is unary, regarding the cost functions we are concerned with, we only need to consider $\mathrm{Cov}(X, X)$, $\mathrm{Cov}(X, X^2)$, and $\mathrm{Cov}(X^2, X^2)$, where $X \sim \mathcal{N}(\mu, \sigma^2)$. Since X is normal, the non-central moments of X can be expressed in terms of μ and σ^2. Hence it is straightforward to compute these covariances [92]. If the operator is binary, then we need to consider $\mathrm{Cov}(X_l, X_l)$, $\mathrm{Cov}(X_r, X_r)$, $\mathrm{Cov}(X_l, X_r)$, $\mathrm{Cov}(X_l X_r, X_l)$, $\mathrm{Cov}(X_l X_r, X_r)$, and $\mathrm{Cov}(X_l X_r, X_l X_r)$. By Lemma 4.5, $X_l \perp X_r$. So we are able to directly compute these covariances as well.

When $i \neq j$, while the types of covariances that we need to consider are similar as before, it is more complicated since the selectivities are no longer independent. Without loss of generality, we consider two operators O and O' such that $O \in \mathrm{Desc}(O')$. By Lemma 4.6, this is the only case where the covariances might not be zero. Based on the cost functions considered in this chapter, we need to consider the covariances $\mathrm{Cov}(Z, Z')$, where $Z \in \{X_l, X_l^2, X_r, X_l X_r\}$ and $Z' \in \{X_l', (X_l')^2, X_r', X_l' X_r'\}$. Some of them can be directly computed by applying Lemma 4.6, while the others can only be bounded as discussed in the next section.

Example 4.14 (Covariances between selectivities)**.** *Consider the two join operators* O_4 *and* O_5 *in Figure 4.1. Assume that the cost functions of* O_4 *and* O_5 *are all linear, i.e., they are of type (C5'). Based on Lemma 4.5,* $\mathrm{Cov}(X_1, X_2) = 0$ *and* $\mathrm{Cov}(X_4, X_3) = 0$*. Also, based on Lemma 4.6,* $\mathrm{Cov}(X_1, X_3) = 0$ *and* $\mathrm{Cov}(X_2, X_3) = 0$*. However, we are not able to compute* $\mathrm{Cov}(X_1, X_4)$ *and* $\mathrm{Cov}(X_2, X_4)$*. Instead, we provide upper bounds for them.*

4.5.3.2 Upper Bounds of Covariances

Based on the fact that the covariance between two random variables is bounded by the geometric mean of their variances [79], we can establish an upper bound for Z

and Z' in the previous section:

$$|\mathrm{Cov}(Z, Z')| \leqslant \sqrt{\mathrm{Var}[Z]\,\mathrm{Var}[Z']}.$$

Note that the variances are directly computable based on the independence assumptions (Lemma 4.5 and 4.6).

By analyzing the correlation of the samples used in selectivity estimation, we can develop tighter bounds (details in Appendix A.3). The key observation here is that the correlations are caused by the samples from the shared relations. Consider two operators O and O' such that $O \in \mathrm{Desc}(O')$. Suppose that $|\mathcal{R} \cap \mathcal{R}'| = m$ ($m \geqslant 1$), namely, O and O' share m common leaf tables. Let the estimators for O and O' be ρ_n and ρ'_n, where n is the number of sample steps. We define $S^2_\rho(m, n)$ to be the variance of samples restricted to the m common relations. This is actually a generalization of $\mathrm{Var}[\rho_n]$. To see this, let $\mathcal{R}' = \mathcal{R}$. Then $\rho_n = \rho'_n$ and hence

$$\mathrm{Var}[\rho_n] = \mathrm{Cov}(\rho_n, \rho_n) = \mathrm{Cov}(\rho_n, \rho'_n) = S^2_\rho(K, n),$$

where $K = |\mathcal{R}|$. We can show that $S^2_\rho(m, n)$ is a monotonically increasing function of m (see Appendix A.3). As a result, $S^2_\rho(m, n) \leqslant \mathrm{Var}[\rho_n]$ given that $m \leqslant K$. Hence, we have the following refined upper bound for $\mathrm{Cov}(\rho_n, \rho'_n)$:

$$|\mathrm{Cov}(\rho_n, \rho'_n)| \leqslant \sqrt{S^2_\rho(m, n) S^2_{\rho'}(m, n)} \leqslant \sqrt{\mathrm{Var}[\rho_n]\,\mathrm{Var}[\rho'_n]}.$$

To compute $S^2_\rho(m, n)$, we use an estimator akin to the estimator $\sigma^2_n = S^2_n/n$ that we used to estimate $\mathrm{Var}[\rho_n]$. Specifically, define

$$S^2_{n,m} = \sum_{r=1}^{m} \left(\frac{1}{n-1} \sum_{j=1}^{n} (Q_{r,j,n}/n^{m-1} - \rho_n)^2 \right),$$

for $n \geqslant 2$ (we set $S^2_{1,m} = 0$). Very similarly, we can show that $\lim_{n\to\infty} S^2_{n,m} = nS^2_\rho(m, n)$. As a result, it is reasonable to approximate $S^2_\rho(m, n)$ with $S^2_\rho(m, n) \approx S^2_{n,m}/n$. Moreover, by comparing the expressions of $S^2_{n,m}$ and S^2_n (ref. Equa-

tion (4.5)), we can see that $S_n^2 = S_{n,K}^2$. Therefore it is straightforward to adapt the implementation framework in Section 4.3.2.2 to compute $S_{n,m}^2$. More discussions on bounding covariances are in Appendix A.4.

4.6 Experimental Evaluation

We present experimental evaluation results in this section. There are two key respects that could impact the utility of a predictor: its prediction accuracy and runtime overhead. However, for the particular purpose of this chapter, we do not care much about the *absolute* accuracy of the prediction. Rather, we care if the distribution of likely running times reflects the uncertainty in the prediction. Specifically, we measure if the estimated prediction errors are correlated with the actual errors. To measure the accuracy of the predicted distribution, we also compare the estimated likelihoods that the actual running times will fall into certain confidence intervals with the actual likelihoods. On the other hand, we measure the runtime overhead of the sampling-based approach in terms of its relative overhead with respect to the original query running time without sampling. We start by presenting the experimental settings and the benchmark queries we used.

4.6.1 Experimental Settings

We implemented our proposed framework in PostgreSQL 9.0.4. We ran PostgreSQL under Linux 3.2.0-26, and we evaluated our approaches with both the TPC-H 1GB and 10 GB databases. Since the original TPC-H database generator uses uniform distributions, to test the effectiveness of the approach under different data distributions, we used the skewed TPC-H database generator [5] to create skewed databases using $z = 1$ (ref. Section 2.5). All experiments were conducted on two machines with the following configurations:

- PC1: Dual Intel 1.86 GHz CPU and 4GB of memory;
- PC2: 8-core 2.40GHz Intel CPU and 16GB of memory.

4.6.2 Benchmark Queries

We created three benchmarks **MICRO**, **SELJOIN**, and **TPCH**:

- **MICRO** consists of pure selection queries (i.e., scans) and two-way join queries. It is a micro-benchmark with the purpose of exploring the strength and weakness of our proposed approach at different points in the selectivity space. We generated the queries with the similar ideas used in the Picasso database query optimizer visualizer [76]. Since the queries have either one (for scans) or two predicates (for joins), the selectivity space is either one or two dimensional. We generated SQL queries that were evenly across the selectivity space, by using the statistics information (e.g., histograms) stored in the database catalogs to compute the selectivities.

- **SELJOIN** consists of selection-join queries with multi-way joins. We generated the queries in the following way. We analyzed each TPC-H query template, and identified the "maximal" sub-query without aggregates. We then randomly generated instance queries from these *reduced* templates. The purpose is to test the particular type of queries to which our proposed approach is tailored — the selection-join queries.

- **TPCH** consists of instance queries from the TPC-H templates. These queries also contain aggregates, and our current strategy is simply ignoring the uncertainty there (recall Section 4.3.2.2). The purpose of this benchmark is to see how this simple work-around works in practice. We used 14 TPC-H templates: 1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 18, and 19. We did not use the other templates since their query plans contain structures that cannot be handled by our current framework (e.g., sub-query plans or views).

We ran each query 5 times and took the average as the actual running time of a query. We cleared both the filesystem cache and the database buffer pool between each run of each query.

4.6.3 Usefulness of Predicted Distributions

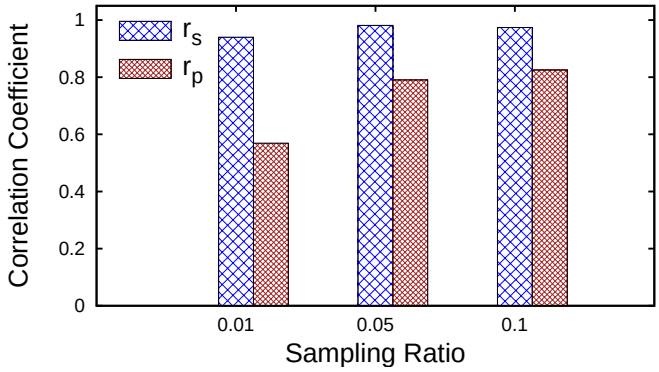
Since our goal is to quantify the uncertainty in the prediction and our output is a distribution of likely running times, the question is then how we can know that we have something useful. A reasonable metric here could be the correlation between the standard deviation of the predicted (normal) distribution and the actual prediction error. Intuitively, the standard deviation indicates the confidence of the prediction. A larger standard deviation indicates lower confidence and hence larger potential prediction error. With this in mind, if our approach is effective, we would expect to see positive correlations between the standard deviations and the real prediction errors when a large number of queries are tested.

A common metric used to measure the correlation between two random variables is the Pearson correlation coefficient r_p. Suppose that we have n queries q_1, ..., q_n. Let σ_i be the standard deviation of the distribution predicted for q_i, μ_i and t_i be the predicted (mean) and actual running time of q_i, and $e_i = |\mu_i - t_i|$ be the prediction error. r_p is then defined as

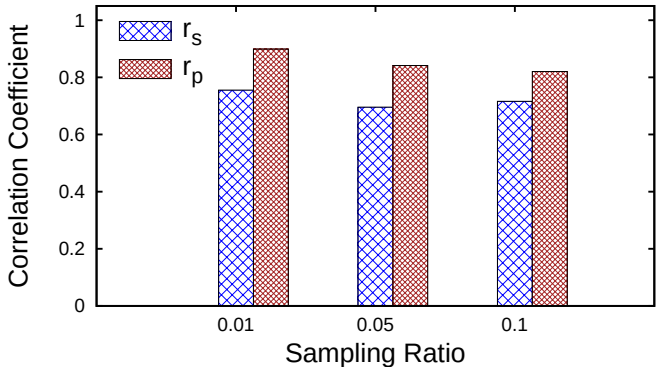
$$r_p = \frac{\sum_{i=1}^{n}(\sigma_i - \bar{\sigma})(e_i - \bar{e})}{\sqrt{\sum_{i=1}^{n}(\sigma_i - \bar{\sigma})^2}\sqrt{\sum_{i=1}^{n}(e_i - \bar{e})^2}}, \tag{4.7}$$

where $\bar{\sigma} = \frac{1}{n}\sum_{i=1}^{n}\sigma_i$ and $\bar{e} = \frac{1}{n}\sum_{i=1}^{n}e_i$.

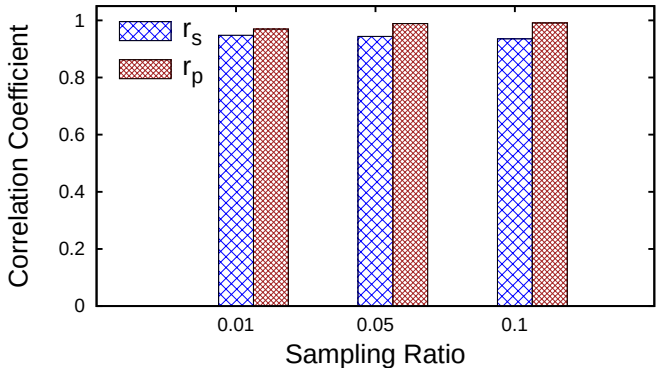
Basically, r_p measures the *linear* correlation between the σ's and the e's. The closer r_p is to 1, the better the correlation is. However, there are two issues here. First, even if the σ's and the e's are positively correlated, the correlation may not be linear. Second, r_p is not robust and its value can be misleading if outliers are present [31]. Therefore, we also measure the correlations by using another well known metric called the Spearman's rank correlation coefficient r_s [69]. The formula of r_s is the same as Equation (4.7) except for that the σ's and e's are replaced with their *ranks* in the ascending order of the values. For instance, given three σ's $\sigma_1 = 4$, $\sigma_2 = 7$, and $\sigma_3 = 5$, their ranks are 1, 3, and 2 respectively. Intuitively, r_s indicates the linear correlation between the ranks of the values, which is more robust than r_p since the mapping from the values to their ranks can be thought of as some



(a) **MICRO**, Uniform 1GB, PC2

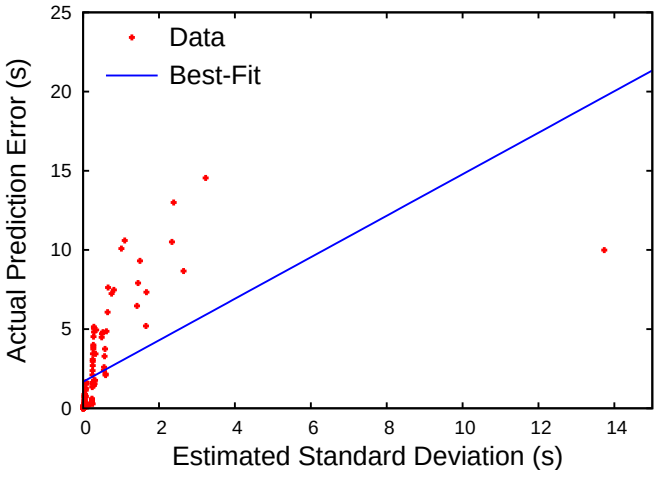


(b) **SELJOIN**, Uniform 1GB, PC1

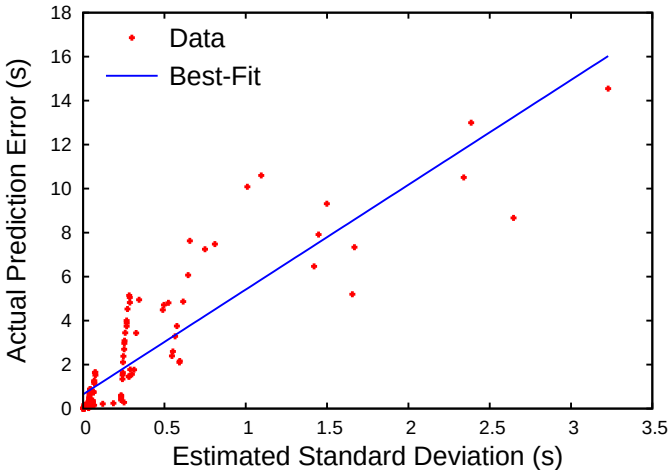


(c) **TPCH**, Skewed 10GB, PC1

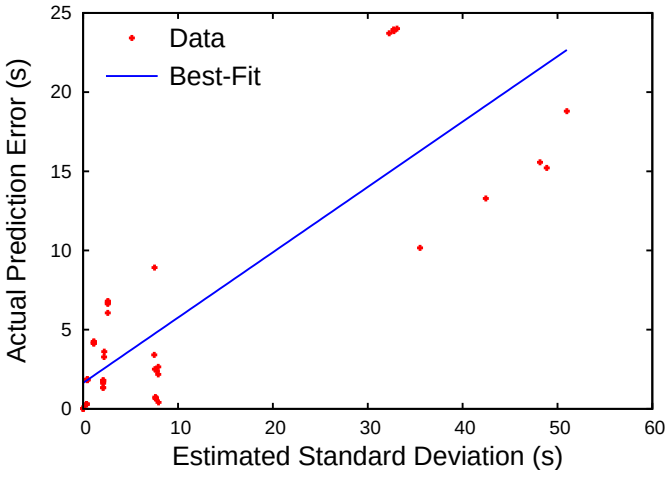
Figure 4.2: r_s and r_p of the benchmark queries over different experimental settings.



(a) Case (1)



(b) Case (1) w/o the outlier



(c) Case (2)

Figure 4.3: Robustness of r_s and r_p with respect to outliers.

normalization procedure that reduces the impact of outliers. In fact, r_s assesses how well the correlation can be characterized by using a *monotonic* function and $r_s = 1$ means the correlation is perfect.

In Figure 4.2, we report the r_s's (and the corresponding r_p's) for the benchmark queries over different experimental settings. Here, sampling ratio (SR) stands for the fraction of the sample size with respect to the database size. For instance, SR = 0.01 means that 1% of the data is taken as samples. We have several observations.

First, for most of the cases we tested, both r_s and r_p are above 0.7 (in fact above 0.9), which implies strong positive (linear) correlation between the standard deviations of the predicted distributions and the actual prediction errors.[6] Second, in Chapter 2 we showed that as expected, prediction errors can be reduced by using larger number of samples. Interestingly, it is not necessarily the case that more samples can improve the correlation between the predicted and actual errors. This is because taking more samples simultaneously reduces the errors in selectivity estimates and the uncertainty in the predicted running times. So it might improve the estimate but not the correlation with the true errors. Third, reporting both r_s and r_p is necessary since they sometimes disagree with each other. For instance, consider the following two cases in Figure 4.2(a) and 4.2(b):

(1) On PC2, the **MICRO** queries over the uniform TPC-H 1GB database give r_s = 0.9400 but r_p = 0.5691 when SR = 0.01;

(2) On PC1, the **SELJOIN** queries over the uniform TPC-H 1GB database give r_s = 0.6958 but r_p = 0.8414 when SR = 0.05.

In Figure 4.3(a) and 4.3(c), we present the scatter plots of these two cases. Figure 4.3(b) further shows the scatter plot after the rightmost point is removed from Figure 4.3(a). We find that now r_s = 0.9386 but r_p = 0.8868. So r_p is much more sensitive to outliers in the population. Since in our context there is no good criterion to remove outliers, r_s is thus more trustworthy. On the other hand, although the

[6] It is generally believed that two variables are strongly correlated if their correlation coefficient is above 0.7.

r_p of (2) is better than that of (1), by comparing Figure 4.3(b) with Figure 4.3(c) we would instead conclude that the correlation of (2) seems to be worse. This is again implied by the worse r_s of (2).

Nonetheless, the strong positive correlations between the estimated standard deviations and the actual prediction errors may not be sufficient to conclude that the distributions of likely running times are useful. For our purpose of informing the consumer of the running time estimates of the potential prediction errors, it might be worth to further consider what information regarding the errors the predicted distributions really carry. Formally, consider the n queries q_1, ..., q_n as before. Since the estimated distributions are normal, with the previous notation the distribution for the likely running times T_i of q_i is $T_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$. As a result, assuming $\alpha > 0$, without loss of generality the estimated prediction error $E_i = |T_i - \mu_i|$ follows the probability distribution

$$\Pr(E_i \leqslant \alpha\sigma_i) = \Pr(-\alpha \leqslant \frac{T_i - \mu_i}{\sigma_i} \leqslant \alpha) = 2\Phi(\alpha) - 1,$$

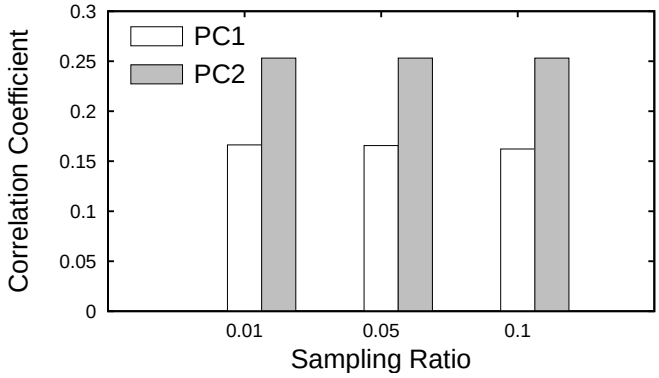
where Φ is the cumulative distribution function of the standard normal distribution $\mathcal{N}(0,1)$. Therefore, if we define the statistic $E_i' = \frac{E_i}{\sigma_i} = |\frac{T_i - \mu_i}{\sigma_i}|$, then $\Pr(E_i' \leqslant \alpha) = \Pr(E_i \leqslant \alpha\sigma_i)$. Note that $\Pr(E_i' \leqslant \alpha)$ is determined by α but not i. We thus simply use $\Pr(\alpha)$ to denote $\Pr(E_i' \leqslant \alpha)$. On the other hand, we can estimate the actual likelihood of $E_i' \leqslant \alpha$ by using

$$\Pr_n(\alpha) = \frac{1}{n}\sum_{i=1}^{n} I(e_i' \leqslant \alpha), \text{ where } e_i' = \frac{e_i}{\sigma_i} = |\frac{t_i - \mu_i}{\sigma_i}|.$$

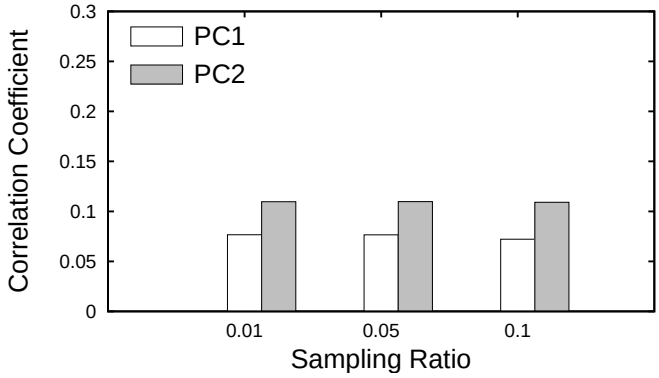
Here I is the indicator function. To measure the proximity of $\Pr_n(\alpha)$ and $\Pr(\alpha)$, we define the following metric

$$D_n(\alpha) = |\Pr_n(\alpha) - \Pr(\alpha)|.$$

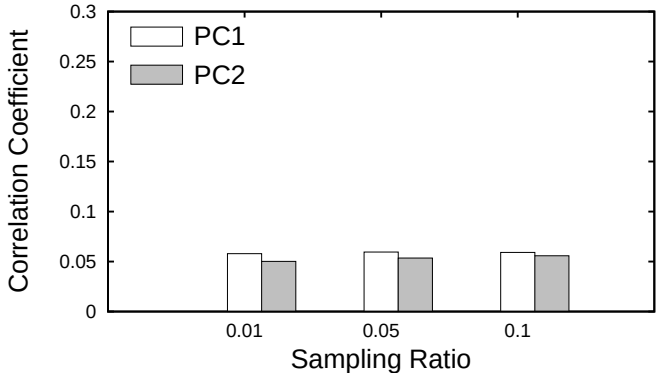
Clearly, a smaller $D_n(\alpha)$ means $\Pr(\alpha)$ is closer to $\Pr_n(\alpha)$, which implies better quality of the distributions. We further generated α's from the interval $(0, 6)$ which



(a) **MICRO**

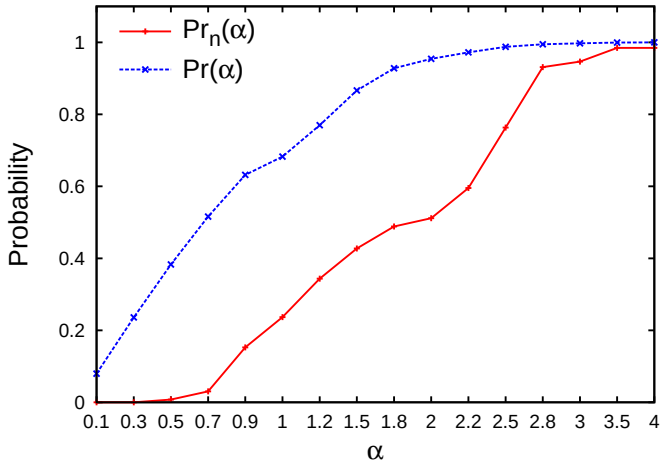


(b) **SELJOIN**

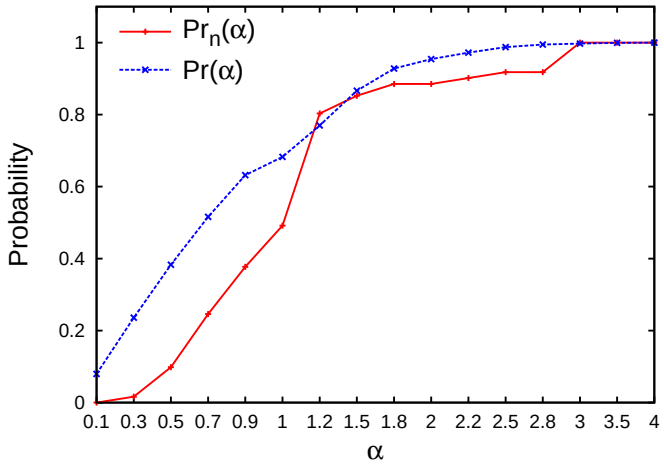


(c) **TPCH**

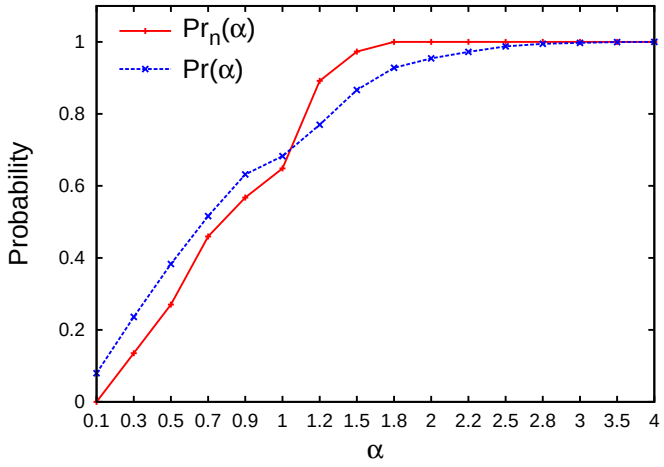
Figure 4.4: $\overline{D}_n$ of the benchmark queries over uniform TPC-H 10GB databases.



(a) Case (1), $\overline{D}_n = 0.2532$



(b) Case (2), $\overline{D}_n = 0.1098$



(c) Case (3), $\overline{D}_n = 0.0535$

Figure 4.5: The proximity of $Pr_n(\alpha)$ and $Pr(\alpha)$ with respect to different $\overline{D}_n$'s.

is sufficiently wide for normal distributions and computed the average of the $D_n(\alpha)$'s (denoted as $\overline{D}_n$). Figure 4.4 reports the results for the benchmark queries over uniform TPC-H 10GB databases.

We observe that in most cases the $\overline{D}_n$'s are below 0.3 with the majority below 0.2, which suggests that the estimated $Pr(\alpha)$'s are reasonably close to the observed $Pr_n(\alpha)$'s. To shed some light on what is going on here, in Figure 4.5 we further plot the $Pr(\alpha)$ and $Pr_n(\alpha)$ for the (1) **MICRO**, (2) **SELJOIN**, and (3) **TPCH** queries over the uniform TPC-H 10GB database on PC2 when SR = 0.05, which give $\overline{D}_n$ = 0.2532, 0.1098, and 0.0535 respectively (see Figure 4.4). We can see that we overestimated the $Pr(\alpha)$'s for small α's. In other words, we underestimated the prediction errors by presenting smaller than actual variances in the distributions. Moreover, we find that overestimate is more significant for the **MICRO** queries (Figure 4.5(a)). One possible reason is that since these queries are really simple the predictor tends to be over-confident by underestimating the variances even more. When handling **SELJOIN** and **TPCH** queries, the confidence of the predictor drops and underestimate tends to be alleviated (Figure 4.5(b) and 4.5(c)).

4.6.4 Runtime Overhead of Sampling

We also measured the relative overhead of running the queries over the sample tables compared with that of running them over the original tables. Figure 4.6 presents the results of the **TPCH** queries on PC1. We observe that the relative overhead is comparable to that reported in Section 2.5. For instance, for the TPC-H 10GB database, the relative overhead is around 0.04 to 0.06 when the sampling ratio is 0.05. Note that, here we computed the estimated selectivities as well as their variances by only increasing the relative overhead a little. Also note that, here we measured the relative overhead based on disk-resident samples. The relative overhead can be dramatically reduced by using the common practice of caching the samples in memory [75].

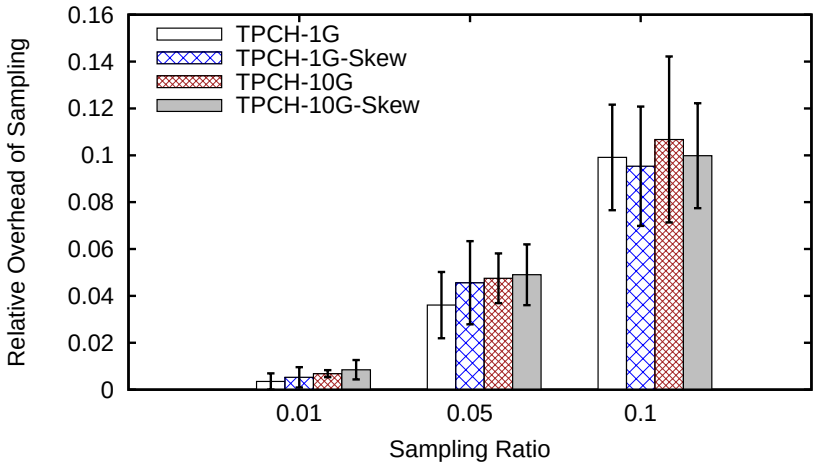


Figure 4.6: Relative overhead of **TPCH** queries on PC1.

4.6.5 Applications

We discuss some potential applications that could take advantage of the distributional information of query running times. The list of applications here is by no means exhaustive, and it is our hope that our study in this chapter could stimulate further research in this direction and more applications could emerge in the future.

4.6.5.1 Query Optimization

Although significant progress has been made in the past several decades, query optimization remains challenging for many queries due to the difficulty in accurately estimating query running times. Rather than betting on the optimality of the plan generated based on (perhaps erroneous) point estimates for parameters such as selectivities and cost units, it makes sense to also consider the uncertainties of these parameters. In fact, there has been some theoretical work investigating optimization based on least *expected* cost (LEC) based upon distributions of the parameters of the cost model [28]. However, that work did not address the problem of how to obtain the distributions. It would be interesting future work to see the effectiveness of LEC plans by incorporating our techniques into query optimizers.

4.6.5.2 Query Progress Monitoring

State-of-the-art query progress indicators [23, 57, 61, 64] provide estimates of the percentage of the work that has been completed by a query at regular intervals during the query's execution. However, it has been shown that in the worst case no progress indictor can outperform a naive indicator simply saying the progress is between 0% and 100% [21]. Hence, information about uncertainty in the estimate of progress is desirable. Our work provides a natural building block that could be used to develop an uncertainty-aware query progress indicator: the progress indicator could call our predictor to make a prediction for the remaining query running time as well as its uncertainty.

4.6.5.3 Database as a Service

The problem of predicting query running time is revitalized by the recent move towards providing database as a service (DaaS). Distributional information enables more robust decision procedures in contrast to point estimates. Recent work [26] has shown the benefits in query scheduling by leveraging distributional information. Similar ideas have also been raised in [96] for admission control. Again, these work did not address the fundamental issue of obtaining the distributions without running the queries. It would be interesting to see the effectiveness of our proposed techniques in these DaaS applications.

4.7 Related Work

The problem of predicting query execution time has been extensively studied quite recently [10, 11, 33, 36, 60, 93, 94]. We have comprehensively discussed this line of work in Chapter 2 and 3.

The idea of using samples to estimate selectivity goes back more than two decades ago (e.g., [14, 19, 43, 44, 45, 49, 50, 62]). While we focused on estimators for selection and join queries [44], some estimators that estimate the number of distinct values might be further used to refine selectivity estimates of aggregate

queries [19, 43]. However, not only do we need an estimate of selectivity, we need an estimated distribution as well. So far, we are not aware of any previous study towards this direction for aggregate queries. Regarding the problem of estimating selectivity distributions for selection and join queries, there are options other than the one used in this chapter. For example, Babcock and Chaudhuri [14] proposed a framework to learn the posterior distributions of the selectivities based on *join synopses* [8]. Unfortunately, this solution is restricted to SPJ (i.e., Select-Project-Join) expressions with foreign-key joins, due to the overhead of computing and maintaining join synopses over a large database.

The framework proposed in this chapter also relies on accurate approximation of the cost models used by the optimizer. Du et al. [32] first proposed the idea of using logical cost functions in the context of heterogenous database systems. Similar ideas were later on used in developing generic cost models for main memory based database systems [65] and identifying robust plans in the plan diagram generated by the optimizer [30]. Our idea of using optimization techniques to find the best coefficients in the logical cost functions is motivated by the approach used in [30].

4.8 Summary

In this chapter, we take a first step towards the problem of measuring the uncertainty within query execution time prediction. We quantify prediction uncertainty using the distribution of likely running times. Our experimental results show that the standard deviations of the distributions estimated by our proposed approaches are strongly correlated with the actual prediction errors.

The idea of leveraging cost models to quantify prediction uncertainty need not be restricted to single standalone queries. The key observation is that the selectivities of the operators in a query are independent of whether or not it is running with other queries (ref. Chapter 3). Hence it is promising to consider applying the techniques proposed in this chapter to multi-query workloads by viewing the interference between queries as changing the distribution of the c's. We regard this as a compelling area for future work.

Chapter 5

Sampling-Based Query Re-Optimization

In the previous chapters, we have investigated the effectiveness of using sampling-based cardinality estimates to get better query running time predictions. Sampling incurs higher overhead and must be used conservatively. Our key observation is the following: while it is infeasible to use sampling for all plans explored by the optimizer during its search for an optimal plan, it is feasible to use sampling as a post-processing step after the search is finished to *detect* potential errors in optimizer's original cardinality estimates for the final chosen plan.

However, if there were really significant errors, the optimality of this final plan would itself be questionable. A natural question is then if sampling could be further used to improve query plans. In this chapter, we study how to exploit the power of sampling in query optimization without significantly increasing optimization time. Built on top of our previous idea of using sampling as a post-processing validation step, we go one step further by introducing a feedback loop from sampling to the query optimizer: if there are significant cardinality estimation errors, we use an iterative procedure to give the optimizer a chance to generate a better plan. We study this re-optimization procedure in detail, both theoretically and experimentally. Our theoretical analysis suggests that the efficiency of this procedure and the quality

of the final plan returned can be guaranteed under certain assumptions, and our experimental evaluation on the TPC-H benchmark as well as our own database with highly correlated data demonstrates the effectiveness of this approach.

5.1 Introduction

Cost-based query optimizers rely on reasonable cost estimates of query plans. Cardinality (or selectivity) estimation is crucial for the accuracy of cost estimates. Unfortunately, although decades of research has been devoted to this area and significant progress has been made, cardinality estimation remains challenging in practice. In current database systems, the dominant approach is to keep various statistics about the data, such as histograms, number of distinct values, and list of most common values (MCV's). While these statistics can work effectively for estimating selectivities of predicates over a single column, they can fail to accurately estimate selectivities for conjunctions of multiple predicates, especially when data correlation is present [63].

Because sampling-based approaches (e.g., [19, 44, 62]) automatically reflect correlation in the data and between multiple predicates over the data, they can provide better cardinality estimates on correlated data than histogram-based approaches. However, sampling also incurs higher overhead and must be used conservatively.

In this chapter, we study how to exploit the power of sampling in query optimization without significantly increasing optimization time. Inspired by the observation that it is feasible to use sampling as a post-processing step to validate cardinality estimates for the final plan returned by the optimizer, our basic idea is simple: if significant cardinality estimation errors are detected, the optimality of the returned plan is then itself questionable, so we go one step further to let the optimizer re-optimize the query by also feeding it the cardinality estimates refined via sampling. This gives the optimizer second chance to generate a different, perhaps better, plan. Note that we can again apply the sampling-based validation step to this new plan returned by the optimizer. It therefore leads to an iterative procedure based on feedback from sampling: we can repeat this optimization-then-validation

loop until the plan chosen by the optimizer does not change. In some sense, our approach can be viewed as a compromise between a histogram-based approach to cardinality estimation and a sampling-based approach — we use sampling to validate cardinality estimates only for a *subset* (rather than all) of the candidate plans considered by the optimizer and choose the final optimal plan from these validated candidates. The hope is that this re-optimization procedure can catch large optimizer errors before the system even begins executing the chosen plan.

A couple of natural concerns arise regarding this simple re-optimzation approach. First, is it efficient? As we have just said, sampling should not be abused given its overhead. Since we propose to run plans over samples iteratively, how fast does this procedure converge? To answer this question, we conduct a theoretical analysis as well as an experimental evaluation. Our theoretical study suggests that, in the worst case, the expected number of iterations can be bounded by $O(\sqrt{N})$, where N is the number of plans considered by the optimizer in its search space. In practice, this can rarely happen, though. Re-optimization for most of the queries tested in our experiments converges after only a few rounds (most likely only one round) of iteration, and the time spent on re-optimization is ignorable compared with the corresponding query running time.

Second, is it useful? Namely, does re-optimization really generate a better plan? In theory, we can prove that, if the cost model used by the optimizer and the sampling-based cardinality estimates are accurate enough, then the re-optimized plan is guaranteed to be no worse than the original plan. However, in practice, cost models are often imperfect, and sampling-based cardinality estimates are imperfect as well. We have implemented our approach in PostgreSQL. Based on experimental results of this implementation, the answer to the above question is: sometimes, but not always. One intriguing example is Q21 of the TPC-H benchmark database. At the scale of 10GB, the running time of the re-optimized plan is about 1 hour while the original plan takes almost 6 hours to finish. For most other TPC-H queries, the re-optimized plans are exactly the same as the original ones. To further evaluate the effectiveness of the re-optimization procedure, we created another database with strong data correlations. Not only PostgreSQL but also

two commercial RDBMS suffer from major query optimization mistakes on this database. Given that optimizers have to handle both common cases and difficult, corner cases, our goal is to help cover some of those corner cases. Our experimental results show the superiority of our approach on this database.

The idea of query re-optimization goes back to nearly two decades ago [56, 66]. The main difference between this line of work and our approach is that re-optimization was previously done *after* a query begins to execute while our re-optimization is done *before* that. While performing re-optimization during the execution of the query has the advantage of being able to observe accurate cardinalities, it suffers from (sometimes significant) runtime overheads such as materializing intermediate results that have been generated in the past period of execution. Meanwhile, runtime re-optimization frameworks usually require significant changes to query optimizer's architecture, such as adding operators that can monitor executions of query plans and adding code that can support switching between query optimization and execution. Our compile-time re-optimization approach is more lightweight. The only additional cost is due to running tentative query plans over samples. The modification to the query optimizer and executor is also limited: our implementation in PostgreSQL needs only several hundred lines of C code. On the other hand, compile-time re-optimization still relies on imperfect cardinality estimates obtained via sampling while runtime re-optimization can benefit from the true cardinalities. Furthermore, we should also note that our compile-time re-optimization approach actually does not conflict with these previous runtime re-optimization techniques: the plan returned by our re-optimization procedure could be further refined by using runtime re-optimization. It remains interesting to investigate the effectiveness of this combination framework.

The rest of this chapter is organized as follows. We present the details of our iterative sampling-based re-optimization algorithm in Section 5.2. We then present a theoretical analysis of its efficiency in terms of the number of iterations it requires and the quality of the final plan it returns in Section 5.3. To evaluate the effectiveness of this approach, we further present a database with highly correlated data in Section 5.4, and we report experimental evaluation results on this database as well

as the TPC-H benchmark databases in Section 5.5. We discuss related work in Section 5.6 and summarize this chapter in Section 5.7.

5.2 The Re-Optimization Algorithm

In this section, we first introduce necessary background information and terminology, and then present the details of the re-optimization algorithm. We focus on using sampling to refine selectivity estimates for join predicates. As was recently pointed out by Guy Lohman [63], while histogram-based approaches have worked well for estimating selectivities of local predicates (i.e., predicates over a column of a single base table), accurate estimation of selectivities for join predicates (and how to combine them) remains challenging. As in previous chapters, the sampling-based selectivity estimator we used is tailored for join queries [44], and it is our goal in this chapter to study its effectiveness in query optimization when combined with our proposed re-optimization procedure. Nonetheless, sampling can also be used to estimate selectivities for other types of operators, such as aggregates (i.e., "group by" clauses) that require estimation of the number of distinct values (e.g. [19]). We leave the exploration of integrating other sampling-based selectivity estimation techniques into query optimization as interesting future work.

5.2.1 Preliminaries

In previous chapters, we used a sampling-based selectivity estimator proposed by Haas et al. [44] for the purpose of predicting query running times. For self-containment purpose, here we provide an informal description of this estimator.

Let R_1, ..., R_K be K relations, and let R_k^s be the sample table of R_k for $1 \leqslant k \leqslant K$. Consider a join query $q = R_1 \bowtie \cdots \bowtie R_K$. The selectivity ρ_q of q is estimated as

$$\hat{\rho}_q = \frac{|R_1^s \bowtie \cdots \bowtie R_K^s|}{|R_1^s| \times \cdots \times |R_K^s|}.$$

It has been shown that this estimator is both unbiased and strongly consistent [44]: the larger the samples are, the more accurate this estimator is. Note that this estimator can be applied to joins that are sub-queries of q as well.

5.2.2 Algorithm Overview

As mentioned in Section 5.1, cardinality estimation is challenging and cardinality estimates by optimizers can be erroneous. This potential error can be noticed once we apply the aforementioned sampling-based estimator to the query plan generated by the optimizer. However, if there are really significant errors in cardinality estimates, the optimality of the plan returned by the optimizer can be in doubt.

If we replace the optimizer's cardinality estimates with sampling-based estimates and ask it to re-optimize the query, what would happen? Clearly, the optimizer would either return the same query plan, or a different one. In the former case, we can just go ahead to execute the query plan: the optimizer does not change plans even with the new cardinalities. In the latter case, the new cardinalities cause the optimizer to change plans. However, this new plan may still not be trustworthy because the optimizer may still decide its optimality based on erroneous cardinality estimates. To see this, let us consider the following example.

Example 5.1. *Consider the two join trees* T_1 *and* T_2 *in Figure 5.1. Suppose that the optimizer first returns* T_1 *as the optimal plan. Sampling-based validation can then collect refined cardinality estimates for the three joins:* $A \bowtie B$, $A \bowtie B \bowtie C$, *and* $A \bowtie B \bowtie C \bowtie D$. *Upon knowing these refined estimates, the optimizer then returns* T_2 *as the optimal plan. However, the join* $C \bowtie D$ *in* T_2 *is not observed in* T_1 *and its cardinality estimate has not been validated yet by using sampling.*

Hence, we can again apply the sampling-based estimator to this new plan and repeat the re-optimization. This leads to an iterative procedure.

Algorithm 5 outlines the above idea. Here, we use Γ to represent the sampling-based cardinality estimates for sub-queries that have been validated by using sampling. Initially, Γ is empty. In the round i ($i \geqslant 1$), the optimizer generates a query

plan P_i based on the current information preserved in Γ (line 5). If P_i is the same as P_{i-1}, then we can terminate the iteration (lines 6 to 8). Otherwise, P_i is new and we invoke the sampling-based estimator over it (line 9). We use Δ_i to represent the sampling-based cardinality estimates for P_i, and we update Γ by merging Δ_i into it (line 10). We then move to the round $i+1$ and repeat the above procedure (line 11).

Algorithm 5: Sampling-based query re-optimization.

Input: q, a given SQL query
Output: P_q, query plan of q after re-optimization

```
1  Γ ← ∅;
2  P_0 ← null;
3  i ← 1;
4  while true do
5  |  P_i ← GetPlanFromOptimizer(Γ);
6  |  if P_i is the same as P_{i-1} then
7  |  |  break;
8  |  end
9  |  Δ_i ← GetCardinalityEstimatesBySampling(P_i);
10 |  Γ ← Γ ∪ Δ_i;
11 |  i ← i + 1;
12 end
13 Let the final plan be P_q;
14 return P_q;
```

In our current implementation of Algorithm 5, we implemented Γ as a hash map where the keys are the joins and the values are the corresponding cardinalities. When the optimizer needs to estimate the cardinality of a join, it first looks up Γ to see if the join is present. If yes, then it simply uses the recorded cardinality. Otherwise, it invokes the histogram-based approaches to estimate the cardinality of the join as usual.

Note that this iterative process has as its goal improving the selected plan, not finding a new globally optimal plan. It is certainly possible that the iterative process misses a good plan because the iterative process does not explore the complete plan space — it only explores neighboring transformations of the chosen plan.

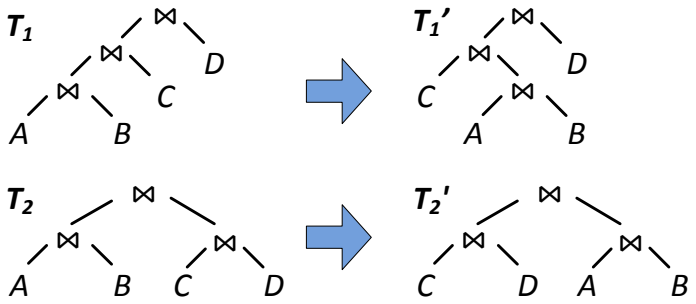


Figure 5.1: Join trees and their local transformations.

Nonetheless, as we will see in Section 5.5, this local search is sufficient to catch and repair some very bad plans.

5.3 Theoretical Analysis

In this section, we present an analysis of Algorithm 5 from a theoretical point of view. We are interested in two aspects of the re-optimization procedure:

- *Efficiency*, i.e., how many rounds of iteration does it require before it ends?
- *Effectiveness*, i.e., how good is the final plan it returns compared to the original plan, in terms of the cost metric used by the query optimizer?

Our following study suggests that (i) the expected number of rounds of iteration in the worst case is upper-bounded by $O(\sqrt{N})$ where N is the number of query plans explored in the optimizer's search space (Section 5.3.3); and (ii) the final plan is guaranteed to be no worse than the original plan if sampling-based cost estimates are consistent with the actual costs (Section 5.3.4).

5.3.1 Local and Global Transformations

We start by introducing the notion of local/global transformations of query plans. In the following, we use $\text{tree}(P)$ to denote the join tree of a query plan P, which is rep-

resented as the *set* of (logical) joins contained in P. For example, the representation of T_1 in Figure 5.1 is $T_1 = \{A \bowtie B, A \bowtie B \bowtie C, A \bowtie B \bowtie C \bowtie D\}$.

Definition 5.2 (Local/Global Transformations)**.** *Two join trees* T *and* T' *are* local *transformations of each other if* T *and* T' *contain the same (logical) joins. Otherwise, they are* global *transformations.*

In other words, local transformations are join trees that subject to only exchanges of left/right subtrees and specific choices of physical join operators (e.g., hash join vs. sort-merge join). In Figure 5.1 we further present two join trees T_1' and T_2' that are local transformations of T_1 and T_2.

Given two plans P and P', we also say that P' is a local/global transformation of P if tree(P') is a local/global transformation of tree(P).

5.3.2 Convergence Conditions

At a first glance, even the convergence of Algorithm 5 is questionable. Is it possible that Algorithm 5 keeps looping without termination? For instance, it seems to be possible that the re-optimization procedure might alternate between two plans P1 and P2, i.e., the plans generated by the optimizer are P1, P2, P1, P2, ... As we will see, this is impossible and Algorithm 5 is guaranteed to terminate. We next present a sufficient condition for the convergence of the re-optimization procedure. We first need one more definition regarding plan coverage.

Definition 5.3 (Plan Coverage)**.** *Given a query plan* P *and a set of query plans* $\mathcal{P}$*,* P *is* covered *by* $\mathcal{P}$ *if*

$$\text{tree}(P) \subseteq \bigcup_{P' \in \mathcal{P}} \text{tree}(P').$$

That is, all the joins in the join tree of P are included in the join trees of $\mathcal{P}$. As a special case, any plan that belongs to $\mathcal{P}$ is covered by $\mathcal{P}$.

Let P_i ($i \geqslant 1$) be the plan returned by the optimizer in the i-th step during the re-optimization procedure.

Theorem 5.4 (Condition of Convergence)**.** *The re-optimization procedure terminates after* $n+1$ *(*$n \geqslant 1$*) steps if* P_n *is covered by* $\mathcal{P} = \{P_1, ..., P_{n-1}\}$*.*

Proof. If P_n is covered by $\mathcal{P}$, then using sampling-based validation will not contribute anything new to the statistics Γ. That is, $\Delta_n \cup \Gamma = \Gamma$. Therefore, P_{n+1} will be the same as P_n, because the optimizer will see the same Γ in the round $n+1$ as that in the round n. Algorithm 5 then terminates accordingly (by lines 6 to 8). □

Note that the convergence condition stated in Theorem 5.4 is sufficient by not necessary. It could happen that P_n is not covered by the previous plans $\{P_1, ..., P_{n-1}\}$ but $P_{n+1} = P_n$ after using the validated statistics (e.g., if there were no significant errors detected in cardinality estimates of P_n).

Corollary 5.5 (Guarantee of Convergence of Re-optimization)**.** *The re-optimization procedure is guaranteed to terminate.*

Proof. Based on Theorem 5.4, the re-optimization procedure would only continue if P_n is not covered by $\mathcal{P} = \{P_1, ..., P_{n-1}\}$. In that case, P_n should contain at least one join that has not been included by the plans in $\mathcal{P}$. Since the total number of possible joins considered by the optimizer is finite, $\mathcal{P}$ will eventually reach some fixed point if it keeps growing. The re-optimization procedure is guaranteed to terminate upon that fixed point, again by Theorem 5.4. □

Theorem 5.4 also implies the following special case:

Corollary 5.6. *The re-optimization procedure terminates after* $n+1$ *(*$n \geqslant 1$*) steps if* $P_n \notin \mathcal{P}$ *but* P_n *is a local transformation of some* $P \in \mathcal{P} = \{P_1, ..., P_{n-1}\}$*.*

Proof. If P_n is a local transformation of some $P \in \mathcal{P}$, then tree(P_n) and tree(P) contain the same joins. By Definition 5.3, P_n is covered by $\mathcal{P}$. Therefore, the re-optimization procedure terminates after $n+1$ steps, by Theorem 5.4. □

Also note that Corollary 5.6 has covered a common case in practice that P_n is a local transformation of P_{n-1}.

Based on Corollary 5.6, in the following we present an important property of the re-optimization procedure.

Theorem 5.7. *When the re-optimization procedure terminates, exactly one of the following three cases holds:*

(1) It terminates after 2 steps with $P_2 = P_1$.

(2) It terminates after $n + 1$ *steps* ($n > 1$). *For* $1 \leqslant i \leqslant n$, P_i *is a global transformation of* P_j *with* $j < i$.

(3) It terminates after $n+1$ *steps* ($n > 1$). *For* $1 \leqslant i \leqslant n-1$, P_i *is a global transformation of* P_j *with* $j < i$, *and* P_n *is a local transformation of some* $P \in \mathcal{P} = \{P_1, ..., P_{n-1}\}$.

Proof. First note that the three cases are mutually exclusive. Next, they are also complete. To see this, note that the only situation not covered is that the re-optimization procedure terminates after $n + 1$ steps ($n > 1$) and during re-optimization there exists some $j < i$ ($1 \leqslant i \leqslant n - 1$) such that P_i is a local transformation of P_j. But this is impossible, because by Corollary 5.6 the re-optimization procedure would then terminate after $i + 1$ steps. Since $i \leqslant n - 1$, we have $i + 1 \leqslant n$, which is contradictory with the assumption that the re-optimization procedure terminates after $n + 1$ steps. Therefore, we have shown the completeness of the three cases stated in Theorem 5.7.

We are left with proving that all these three cases could happen when the re-optimization procedure terminates. Case (1) and (2) are clearly possible, while Case (3) is implied by Corollary 5.6. This completes the proof of the theorem. □

That is, when the procedure does not terminate trivially (Case (1)), it can be characterized as a sequence of global transformations with *at most* one more local transformation before its termination (Case (2) or (3)). Figure 5.2 illustrates the latter two nontrivial cases.

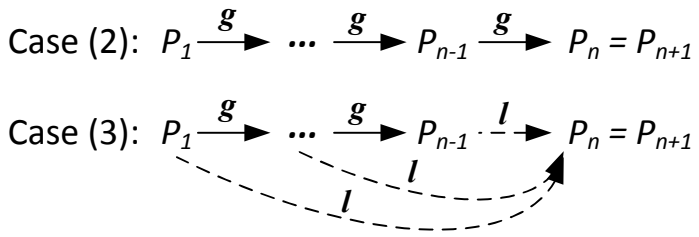


Figure 5.2: Characterization of the re-optimization procedure (g and l stand for global and local transformations, respectively). For ease of illustration, P_i is only noted as a global transformation of P_{i-1}, but we should keep in mind that P_i is also a global transformation of all the P_j's with $j < i$.

5.3.3 Efficiency

We are interested in how fast the re-optimization procedure terminates. As pointed out by Theorem 5.7, the convergence speed depends on the number of *global* transformations the procedure undergoes. In the following, we first develop a probabilistic model that will be used in our analysis, and then present analytic results for the general case and several important special cases.

5.3.3.1 A Probabilistic Model

Consider a queue of N balls. Originally all balls are not marked. We then conduct the following procedure:

Procedure 1. *In each step, we pick the ball at the head of the queue. If it is marked, then the procedure terminates. Otherwise, we mark it and randomly insert it back into the queue: the probability that the ball will be inserted at the position* i $(1 \leqslant i \leqslant N)$ *is uniformly* $1/N$.

We next study the expected number of steps that Procedure 1 would take before its termination.

Lemma 5.8. *The expected number of steps that Procedure 1 takes before its termination is:*

$$S_N = \sum_{k=1}^{N} k \cdot (1 - \frac{1}{N}) \cdots (1 - \frac{k-1}{N}) \cdot \frac{k}{N}. \tag{5.1}$$

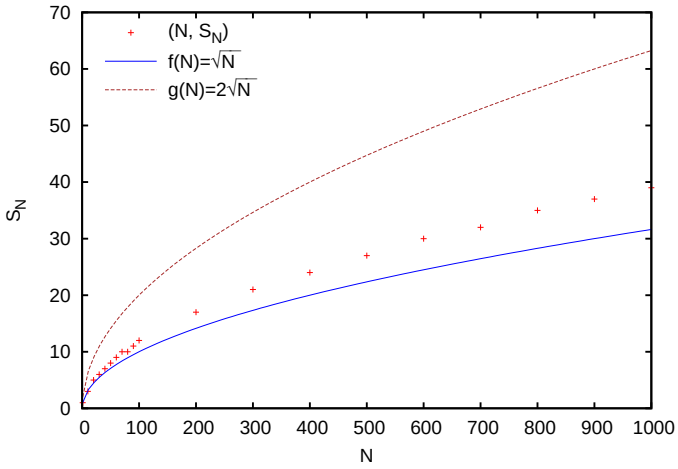


Figure 5.3: S_N with respect to the growth of N.

We include a proof of Lemma 5.8 in Appendix A.5. We can further show that S_N is upper-bounded by $O(\sqrt{N})$.

Theorem 5.9. *The expected number of steps* S_N *as presented in Lemma 5.8 satisfies*

$$S_N = O(\sqrt{N}).$$

The proof of Theorem 5.9 is in Appendix A.6. In Figure 5.3, we plot S_N by increasing N from 1 to 1000. As we can see, the growth speed of S_N is very close to that of $f(N) = \sqrt{N}$.

5.3.3.2 The General Case

In a nutshell, query optimization can be thought of as picking a plan with the lowest estimated cost among a number of candidates. Different query optimizers have different search spaces, so in general we can only assume a search space with N different join trees that will be considered by an optimizer.[1] Let these trees be T_1, ..., T_N, ordered by their estimated costs. The re-optimization procedure can then

[1]By "different" join trees, we mean join trees that are global transformations of each other. We use this convention in the rest of this chapter unless specific clarifications are noted.

be thought of as shuffling these trees based on their refined cost estimates. This procedure terminates whenever (or even before) the tree with lowest estimated cost reoccurs, that is, when some tree appears at the head position for the second time. Therefore, the probabilistic model in the previous section applies here. As a result, by Lemma 5.8, the expected number of steps for this procedure to terminate is S_N. We formalize this result as the following theorem:

Theorem 5.10. *Assume that the position of a plan (after sampling-based validation) in the ordered plans with respect to their costs is uniformly distributed. Let* N *be the number of different join trees in the search space. The expected number of steps before the re-optimization procedure terminates is then* S_N*, where* S_N *is computed by Equation (5.1). Moreover,* $S_N = O(\sqrt{N})$ *by Theorem 5.9.*

We emphasize that the analysis here only targets worst-case performance, which might be too pessimistic. This is because Procedure 1 only simulates the Case (3) stated in Theorem 5.7, which is the worst one among the three possible cases. In our experiments, we have found that all queries we tested require less than 10 rounds of iteration, most of which require only 1 or 2 rounds.

Remark The uniformity assumption in Theorem 5.10 may not be valid in practice. It is possible that a plan after sampling-based validation (or, a marked ball in terms of Procedure 1) is more likely to be inserted into the front/back half of the queue. Such cases imply that, rather than with an equal chance of overestimation/underestimation, the optimizer tends to overestimate/underestimate the costs of all query plans (for a particular query). This is, however, not impossible. In practice, significant cardinality estimation errors usually appear locally and propagate upwards. Once the error at some join were corrected, the errors in all plans that contain that join would also be corrected. In other words, the correction of the error at a single join can lead to the correction of errors in many candidate plans. In Appendix A.7, we further present analysis for two extreme cases: all local errors are overestimates/underestimates. Any real case sits in between. To summarize, for left-deep join trees, we have the following two results:

- If all local errors are overestimates, then in the worst case the re-optimization procedure will terminate in at most $m + 1$ steps, where m is the number of joins contained in the query.
- If all local errors are underestimates, then in the worst case the re-optimization procedure is expected to terminate in $S_{N/M}$ steps, where N is the number of different join trees in the optimizer's search space and M is the number of edges in the join graph.

Note that both results are better than the bound stated in Theorem 5.10. For instance, in the underestimation-only case, if $N = 1000$ and $M = 10$, we have $S_N = 39$ but $S_{N/M} = 12$.

5.3.4 Optimality of the Final Plan

We can think of the re-optimization procedure as progressive adjustments of the optimizer's direction when it explores its search space. Of course, the search space depends on the algorithm or strategy used by the optimizer, and thus the impact of re-optimization also depends on that. But we can still have some general conclusions about the optimality of the final plan regardless of the search space.

Assumption 2. *The cost estimates of plans after using sampling-based cardinality refinement are consistent. That is, for any two plans* P_1 *and* P_2*, if* $\text{cost}^s(P_1) < \text{cost}^s(P_2)$*, then* $\text{cost}^a(P_1) < \text{cost}^a(P_2)$*. Here,* $\text{cost}^s(P)$ *and* $\text{cost}^a(P)$ *are the estimated cost based on sampling and the actual cost of plan* P*, respectively.*

We have the following theorem based on Assumption 2.

Theorem 5.11. *Let* P_1*, ...,* P_n *be a series of plans generated during re-optimization. Then* $\text{cost}^s(P_n) \leqslant \text{cost}^s(P_i)$*, and thus, by Assumption 2, it follows that* $\text{cost}^a(P_n) \leqslant \text{cost}^a(P_i)$*, for* $1 \leqslant i \leqslant n-1$.

Proof. First note that $\text{cost}^s(P_i)$ is well defined for $1 \leqslant i \leqslant n$, because all these plans have been validated via sampling. As a result, all the statistics regarding P_1, ...,

P_{n-1} were already included in Γ when the re-optimization procedure returned P_n as the optimal plan. Since P_n is the final plan, it implies that it did not change in the final round of iteration, where Γ also included statistics of P_n as well. Then the only reason for the optimizer to pick P_n as the optimal plan is $\text{cost}^s(P_n) \leqslant \text{cost}^s(P_i)$ $(1 \leqslant i \leqslant n-1)$. □

That is, the plan after re-optimization is guaranteed to be better than the original plan. Nonetheless, it is difficult to conclude that the plans are improved monotonically, namely, in general it is not true that $\text{cost}^s(P_{i+1}) \leqslant \text{cost}^s(P_i)$, for $1 \leqslant i \leqslant n-1$. However, we can prove that this is true if we only have overestimates.

Corollary 5.12. *Let* P_1, ..., P_n *be a series of plans generated during re-optimization. If in the re-optimization procedure only overestimates occur, then* $\text{cost}^s(P_{i+1}) \leqslant \text{cost}^s(P_i)$ *for* $1 \leqslant i \leqslant n-1$.

Proof. The argument is similar to that used in the proof of Theorem 5.11. When the optimizer returned P_{i+1} as the optimal plan, it had already seen the statistics of P_i after sampling. The only reason it chose P_{i+1} is then $\text{cost}^o(P_{i+1}) \leqslant \text{cost}^s(P_i)$. Here $\text{cost}^o(P_{i+1})$ is the original cost estimate of P_{i+1} (perhaps based on histograms), because at this point sampling has not been used for P_{i+1} yet. But given that only overestimation is possible during re-optimization, it follows that $\text{cost}^s(P_{i+1}) \leqslant \text{cost}^o(P_{i+1})$. As a result, $\text{cost}^s(P_{i+1}) \leqslant \text{cost}^s(P_i)$. Note that i is arbitrary in the argument so far. Thus we have proved the theorem. □

Our last result on the optimality of the final plan is in the sense that it is the best among all the plans that are local transformations of the final plan.

Theorem 5.13. *Let* P *be the final plan returned by the re-optimization procedure. For any* P' *such that* P' *is a local transformation of* P*, it holds that* $\text{cost}^s(P) \leqslant \text{cost}^s(P')$.

Proof. The proof is straightforward. By Theorem 5.7, local transformation is only possible in the last second step of re-optimization. At this stage, Γ has already included all statistics regarding P and P': actually they share the same joins given that they are local transformations. Therefore, $\text{cost}^s(P)$ and $\text{cost}^s(P')$ are both

known to the optimizer. Then $\text{cost}^s(P) \leqslant \text{cost}^s(P')$ must hold because the optimizer decides that P, rather than P', is the optimal plan. □

5.3.5 Discussion

We call the final plan returned by the re-optimization procedure the *fixed point* with respect to the initial plan generated by the optimizer. According to Theorem 5.11, this plan is a *local optimum* with respect to the initial plan. Note that, if $\mathcal{P} = \{P_1, ..., P_n\}$ covers the whole search space, that is, any plan P in the search space is covered by $\mathcal{P}$, then the locally optimal plan obtained is also globally optimal.

A natural question is then the impact of the initial plan. Intuitively, it seems that the initial plan can affect both the fixed point and the time it takes to converge to the fixed point. (Note that it is straightforward to prove that the fixed point must exist and be unique, with respect to the given initial plan.) There are also other related interesting questions. For example, if we start with two initial plans with similar cost estimates, would they converge to fixed points with similar costs as well? We leave all these problems as interesting directions for further investigation.

Moreover, the convergence of the re-optimization procedure towards a fixed point can also be viewed as a validation procedure of the costs of the plans $\mathcal{V}$ that can be covered by $\mathcal{P} = \{P_1, ..., P_n\}$. Note that $\mathcal{V}$ is a subset of the whole search space explored by the optimizer, and $\mathcal{V}$ is induced by P_1 — the initial plan that is deemed as optimal by the optimizer. It is also interesting future work to study the relationship between P_1 and $\mathcal{V}$, especially how much of the complete search space can be covered by $\mathcal{V}$.

5.4 Optimizer "Torture Test"

Evaluating the effectiveness of a query optimizer is challenging. As we mentioned in Section 5.1, query optimizers have to handle not only common cases but also difficult, corner cases. In view of this, we conduct performance evaluation for both common cases and corner cases. For the former, we use the TPC-H benchmark

databases. For the latter, we focus on queries whose cardinalities are difficult to be accurately estimated by using histogram-based approaches. Specifically, we generate a special database as well as a special set of queries with controlled degree of data correlation. We call it "optimizer torture test" (OTT), given that our goal is to sufficiently challenge the cardinality estimation approaches used by current query optimizers. We next describe the details of OTT.

5.4.1 Design of the Database and Queries

Since we target cardinality/selectivity estimation, we can focus on queries that only contain selections and joins. In general, a selection-join query q over K relations R_1, ..., R_K can be represented as

$$q = \sigma_F(R_1 \bowtie \cdots \bowtie R_K),$$

where F is a selection predicate as in relational algebra (i.e., a boolean formula). Moreover, we can just focus on equality predicates, i.e., predicates of the form $A = c$ where A is an attribute and c is a constant. Any other predicate can be represented by unions of equality predicates. As a result, we can focus on F of the form

$$F = (A_1 = c_1) \wedge \cdots \wedge (A_K = c_K),$$

where A_k is an attribute of R_k, and $c_k \in \mathrm{Dom}(A_k)$ $(1 \leqslant k \leqslant K)$. Here, $\mathrm{Dom}(A_k)$ is the domain of the attribute A_k.

Based on these observations, our design of the database and queries is as follows:

(1) We have K relations $R_1(A_1, B_1)$, ..., $R_K(A_K, B_K)$.

(2) We use A_k's for selections and B_k's for joins.

(3) Let $R'_k = \sigma_{A_k = c_k}(R_k)$ for $1 \leqslant k \leqslant K$. The queries of our benchmark are then of the following form:

$$R'_1 \bowtie_{B_1 = B_2} R'_2 \bowtie_{B_2 = B_3} \cdots \bowtie_{B_{K-1} = B_K} R'_K. \tag{5.2}$$

The remaining question is how to generate data for R_1, ..., R_K so that we can easily control the selectivities for the selection and join predicates. This requires us to consider the joint data distribution for $(A_1, ..., A_K, B_1, ..., B_K)$. A straightforward way could be to specify the contingency table of the distribution. However, there is a subtle issue of this approach: we cannot just generate a large table with attributes A_1, ..., A_K, B_1, ..., B_K and then split it into different relations $R_1(A_1, B_1)$, ..., $R_K(A_K, B_K)$. To see why this is incorrect, let us consider the following example.

Example 5.14 (Approach Using Joint Distribution)**.** *Suppose that we need to generate binary values for two attributes A_1 and A_2, with the following joint distribution:*

(1) $p_{00} = \Pr(A_1 = 0, A_2 = 0) = 0.1$;

(2) $p_{01} = \Pr(A_1 = 0, A_2 = 1) = 0$;

(3) $p_{10} = \Pr(A_1 = 1, A_2 = 0) = 0$;

(4) $p_{11} = \Pr(A_1 = 1, A_2 = 1) = 0.9$.

Assume that we generate 10 tuples (A_1, A_2) according to the above distribution. Then we would expect to obtain a multiset of tuples: $\{1 \cdot (0,0), 9 \cdot (1,1)\}$. Here the notation $1 \cdot (0,0)$ means there is one $(0,0)$ in the multiset. If we project the tuples onto A_1 and A_2 (without removing duplicates), we have $A_1 = A_2 = \{1 \cdot 0, 9 \cdot 1\}$. Now what is the joint distribution of (A_1, A_2) once we see such a database? Note that we have no idea about the true *joint distribution that governs the generation of the data, because we are only allowed to see the* marginal *distributions of A_1 and A_2. A natural inference of the joint distribution could be to consider the cross product $A_1 \times A_2$. In this case we have*

$$\begin{aligned} A_1 \times A_2 &= \{1 \cdot 0, 9 \cdot 1\} \times \{1 \cdot 0, 9 \cdot 1\} \\ &= \{1 \cdot (0,0), 9 \cdot (0,1), 9 \cdot (1,0), 81 \cdot (1,1)\}. \end{aligned}$$

Hence, the "observed" joint distribution of A_1 and A_2 is:

(1) $p'_{00} = \Pr'(A_1 = 0, A_2 = 0) = 1/100 = 0.01$;

(2) $p'_{01} = \Pr'(A_1 = 0, A_2 = 1) = 9/100 = 0.09$;

(3) $p'_{10} = \Pr'(A_1 = 1, A_2 = 0) = 9/100 = 0.09$;

(4) $p'_{11} = \Pr'(A_1 = 1, A_2 = 1) = 81/100 = 0.81$.

The "observed" joint distribution is indeed "the" joint distribution when tables are joined. It might be easier to see this if we rewrite the query in Equation (5.2) as

$$\sigma_{A_1=c_1 \wedge \cdots \wedge A_K=c_K \wedge B_1=B_2 \wedge \cdots \wedge B_{K-1}=B_K}(R_1 \times \cdots \times R_K). \tag{5.3}$$

The previous example shows that there is information loss when marginalizing out attributes, which is somewhat similar to the notion of lossless/lossy joins in database normalization theory. This discrepancy between the true and observed joint distributions calls for a new approach.

5.4.2 The Data Generation Algorithm

The previous analysis suggests that we can only generate data for each $R_k(A_k, B_k)$ separately and independently, without resorting to their joint distribution. To generate correlated data, we therefore have to make A_k and B_k correlated, for $1 \leqslant k \leqslant K$. Because our goal is to challenge the optimizer's cardinality estimation algorithm, we choose to go to the extreme of this direction: let B_k be the same as A_k. Algorithm 6 presents the details of this idea.

Algorithm 6: Data generation for the OTT database.

Input: $\Pr(A_k)$, the distribution of A_k, for $1 \leqslant k \leqslant K$
Output: $R_k(A_k, B_k)$, tables generated, for $1 \leqslant k \leqslant K$

```
for 1 ⩽ k ⩽ K do
    Pick a seed independently for the random number generator;
    Generate A_k with respect to Pr(A_k);
    Generate B_k = A_k;
end
return R_k(A_k, B_k), 1 ⩽ k ⩽ K;
```

We are left with the problem of specifying $\Pr(A_k)$. While $\Pr(A_k)$ could be arbitrary, we should reconsider our goal of sufficiently challenging the optimizer. We therefore need to know some details about how the optimizer estimates selectivities/cardinalities. Of course, different query optimizers have different implementations, but the general principles are similar. In the following, we present the specific technique used by PostgreSQL, which is used in our evaluation in Section 5.5.

5.4.2.1 PostgreSQL's Approaches

PostgreSQL stores the following three types of statistics for each attribute A in its `pg_stats` view [4], if the `ANALYZE` command is invoked for the database:

- the number of distinct values $n(A)$ of A;
- a list of most common values (MCV's) of A and their frequency;
- an equal-depth histogram for the other values of A except for the MCV's.

The above statistics can be used to estimate the selectivity of a predicate over a single attribute in a straightforward manner. For instance, for the predicate $A = c$ in our OTT queries, PostgreSQL first checks if c is in the MCV's. If c is present, then the optimizer simply uses the (exact) frequency recorded. Otherwise, the optimizer assumes a uniform distribution over the non-MCV's and estimates the frequency of c based on $n(A)$.

The approach used by PostgreSQL to estimate selectivities for join predicates is more sophisticated. Consider an equal-join predicate $B_1 = B_2$. If MCV's for either B_1 or B_2 are not available, then the optimizer uses an approach first introduced in System R [82] by estimating the reduction factor as $1/\max\{n(B_1), n(B_2)\}$. If, on the other hand, MCV's are available for both B_1 and B_2, then PostgreSQL tries to refine its estimate by first "joining" the two lists of MCV's. For skewed data distributions, this can lead to much better estimates because the join size of the MCV's, which is accurate, would be very close to the actual join size. Other database systems, such as Oracle [16], use similar approaches.

To combine selectivities from multiple predicates, PostgreSQL relies on the attribute-value-independence (AVI) assumption, which assumes that the distributions of values of different attributes are independent.

5.4.2.2 The Distribution $\Pr(A_k)$ And Its Impact

From the previous discussion we can see that whether $\Pr(A_k)$ is uniform or skewed would have little difference in affecting the optimizer's estimates if MCV's were leveraged, simply because MCV's have recorded the *exact* frequency for those skewed values. We therefore can just let $\Pr(A_k)$ be uniform. We next analyze the impact of this decision by computing the differences between the estimated and actual cardinalities for the OTT queries.

Let us first revisit the OTT queries presented in Equation (5.2). Note that for an OTT query to be non-empty, the following condition must hold:

$$B_1 = B_2 = \cdots = B_{K-1} = B_K.$$

Because we have intentionally set $A_k = B_k$ for $1 \leqslant k \leqslant K$, this then implies

$$A_1 = A_2 = \cdots = A_{K-1} = A_K. \tag{5.4}$$

The query size can thus be controlled by the values of the A's. The query is simply empty if Equation (5.4) does not hold. We now compute the query size when Equation (5.4) holds.

Consider a specific query q where the constants in the selection predicates are fixed. Let us compute the cardinality of q, which is equivalent to computing the selectivity of the predicate in Equation (5.3). In probabilistic sense, the selectivity s can be interpreted as the chance that a (randomly picked) tuple from $R_1 \times \cdots \times R_K$ making the selection predicate true. That is,

$$s = \Pr(A_1 = c_1, \cdots, A_K = c_K, B_1 = \cdots = B_K).$$

Lemma 5.15. *When Equation (5.4) holds, we have*

$$s = \prod_{k=1}^{K} \frac{1}{n(A_k)}.$$

Proof. We have $s = \Pr(A_1 = c_1, \cdots, A_K = c_K, B_1 = \cdots = B_K)$, or equivalently,

$$\begin{aligned} s &= \Pr(B_1 = \cdots = B_K | A_1 = c_1, \cdots, A_K = c_K) \\ &\times \Pr(A_1 = c_1, \cdots, A_K = c_K). \end{aligned}$$

For notational convenience, define

$$\Pr(\mathbf{B}|\mathbf{A}) = \Pr(B_1 = \cdots = B_K | A_1 = c_1, \cdots, A_K = c_K),$$

and similarly define

$$\Pr(\mathbf{A}) = \Pr(A_1 = c_1, \cdots, A_K = c_K).$$

Since $B_k = A_k$ $(1 \leqslant k \leqslant K)$, $\Pr(\mathbf{B}|\mathbf{A}) = 1$ when Equation (5.4) holds. Therefore, $s = \Pr(\mathbf{A})$. Moreover, since we generate R_k independently and $\Pr(A_k)$ is uniform $(1 \leqslant k \leqslant K)$, it then follows that

$$s = \Pr(\mathbf{A}) = \prod_{k=1}^{K} \Pr(A_k = c_k) = \prod_{k=1}^{K} \frac{1}{n(A_k)},$$

where $n(A_k)$ is the number of distinct values of A_k as before. This completes the proof of the lemma. □

As a result, the size of the query q is

$$|q| = s \cdot \prod_{k=1}^{K} |R_k| = \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}.$$

To summarize, we have

$$|q| = \begin{cases} \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}, & \text{if } c_1 = \cdots = c_K; \\ 0, & \text{otherwise.} \end{cases}$$

Now what would be the query size estimated by the optimizer? Again, let us compute the estimated selectivity $\hat{s}$, assuming that the optimizer knows the exact histograms of the A_k and B_k ($1 \leqslant k \leqslant K$). Note that this assumption is stronger than the case in Section 5.4.2.1, where the optimizer possesses exact histograms only for MCV's. We then have the following result:

Lemma 5.16. *Suppose that the AVI assumption is used. Assuming that* $\mathrm{Dom}(B_k)$ *is the same for* $1 \leqslant k \leqslant K$ *and* $|\mathrm{Dom}(B_k)| = L$, *we have*

$$\hat{s} = \frac{1}{L^{K-1}} \prod_{k=1}^{K} \frac{1}{n(A_k)}.$$

Proof. Assume that $\mathrm{Dom}(B_k) = \{C_1, ..., C_L\}$ for $1 \leqslant k \leqslant K$. We have $\hat{s} = \Pr(A_1 = c_1, \cdots, A_K = c_K, B_1 = B_2, \cdots, B_{K-1} = B_K)$. According to the AVI assumption, it follows that

$$\hat{s} = \prod_{k=1}^{K} \Pr(A_k = c_k) \times \prod_{k=1}^{K-1} \Pr(B_k = B_{k+1}).$$

Let us consider $\Pr(B_k = B_{k+1})$. We have

$$\begin{aligned} \Pr(B_k = B_{k+1}) &= \sum_{l=1}^{L} \sum_{l'=1}^{L} \big(\Pr(B_k = C_l, B_{k+1} = C_{l'}) \\ &\times \Pr(B_k = B_{k+1} | B_k = C_l, B_{k+1} = C_{l'}) \big). \end{aligned}$$

Since $\Pr(B_k = B_{k+1} | B_k = C_l, B_{k+1} = C_{l'}) = 1$ if and only if $l = l'$ (otherwise it equals 0), it follows that

$$\Pr(B_k = B_{k+1}) = \sum_{l=1}^{L} \Pr(B_k = C_l, B_{k+1} = C_l).$$

Moreover, since we have assumed that the optimizer knows exact histograms and $\Pr(B_k)$ is uniform for $1 \leqslant k \leqslant K$ (recall that $B_k = A_k$ and $\Pr(A_k)$ is uniform),

$$\begin{aligned}\Pr(B_k = C_l, B_{k+1} = C_l) &= \frac{n(B_k = C_l, B_{k+1} = C_l)}{|B_k| \cdot |B_{k+1}|} \\ &= \frac{n(B_k = C_l)}{|B_k|} \cdot \frac{n(B_{k+1} = C_l)}{|B_{k+1}|} \\ &= \frac{1}{n(B_k) \cdot n(B_{k+1})}.\end{aligned}$$

Because $n(B_k) = L$ for $1 \leqslant k \leqslant K$, it follows that

$$\Pr(B_k = B_{k+1}) = L \times \frac{1}{L^2} = \frac{1}{L},$$

and as a result,

$$\hat{s} = \frac{1}{L^{K-1}} \prod_{k=1}^{K} \Pr(A_k = c_k) = \frac{1}{L^{K-1}} \prod_{k=1}^{K} \frac{1}{n(A_k)}.$$

This completes the proof of the lemma. □

Hence, the estimated size of the query q is

$$\widehat{|q|} = \hat{s} \cdot \prod_{k=1}^{K} |R_k| = \frac{1}{L^{K-1}} \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}.$$

Note that this is regardless of if Equation (5.4) holds or not. In our experiments (Section 5.5) we used this property to generate instance OTT queries.

Comparing the expressions of $|q|$ and $\widehat{|q|}$, if we define

$$d = \left| |q| - \widehat{|q|} \right|,$$

it then follows that

$$d = \begin{cases} \left(1 - \frac{1}{L^{K-1}}\right) \cdot \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}, & \text{if } c_1 = \cdots = c_K; \\ \frac{1}{L^{K-1}} \cdot \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}, & \text{otherwise.} \end{cases}$$

Let us further get some intuition about how large d could be by considering the following specific example.

Example 5.17. *For simplicity, assume that* $M = |R_k|/n(A_k)$ *is the same for* $1 \leqslant k \leqslant K$. M *is then the number of tuples per distinct value of* A_k *(*$1 \leqslant k \leqslant K$*). In this case,*

$$d = \begin{cases} \left(1 - 1/L^{K-1}\right) \cdot M^K, & \text{if } c_1 = \cdots = c_K; \\ M^K/L^{K-1}, & \text{otherwise.} \end{cases}$$

If $L = 100$ *and* $M = 100$*, it then follows that*

$$d = \begin{cases} 100^K - 100, & \text{if } c_1 = \cdots = c_K; \\ 100, & \text{otherwise.} \end{cases}$$

For $K = 4$ *(i.e., just 3 joins),* $d \approx 10^8$ *if it happens to be that* $c_1 = \cdots = c_K$*. Moreover, if* $c_1 = \cdots = c_K$*, then* $|q| > \widehat{|q|}$*. Therefore, the optimizer would significantly underestimate the size of the join (by* 10^8*). So it is likely that the optimizer would pick an inefficient plan that it thought were efficient.*

5.5 Experimental Evaluation

We present experimental evaluation results of our proposed re-optimization procedure in this section.

5.5.1 Experimental Settings

We implemented our proposed re-optimization framework in PostgreSQL 9.0.4. The modification to the optimizer is small, limited to several hundred lines of C code, which demonstrates the feasibility of including our framework into current query optimizers. We conducted our experiments on a PC configured with 2.4GHz Intel dual-core CPU and 4GB memory, and we ran PostgreSQL under Linux 2.6.18.

5.5.1.1 Databases and Performance Metrics

We used both the standard version and a skewed version [5] of the TPC-H benchmark database, as well as our own OTT database described in Section 5.4.

TPC-H Benchmark Databases We used 10GB TPC-H databases in our experiments. As before, we generated a skewed database by setting $z = 1$.

OTT Database We created a specific instance of the OTT database in the following manner. We first generated a standard 1GB TPC-H database. We then extended the 6 largest tables (*lineitem*, *orders*, *partsupp*, *part*, *customer*, *supplier*) by adding two additional columns A and B to each of them. As discussed in Section 5.4.2, we populated the extra columns with uniform data. The domain of a column is determined by the number of rows in the corresponding table: if the table contains r rows, then the domain is $\{0, 1, ..., r/100 - 1\}$. In other words, each distinct value in the domain would appear roughly 100 times in the generated column. We further created an index on each added column.

Performance Metrics We measured the following performance metrics for each query on each database:

(1) the original running time of the query;

(2) the number of iterations the re-optimization procedure requires;

(3) the time spent on the re-optimization procedure;

(4) the total running time of the query including the re-optimization time.

Based on studies in previous chapters, in all of our experiments we set the sampling ratio to be 0.05, namely, 5% of the data were taken as samples.

5.5.1.2 Calibrating Cost Models

In Chapter 2, we have also shown that, after proper calibration of the cost models used by the optimizer, we could have better estimates of query running times. An interesting question is then: would calibration also help with query optimization?

In our experiments, we also investigated this problem. Specifically, we ran the offline calibration procedure (details in Section 2.3) and replaced the default values of the five cost units in `postgresql.conf` (i.e., the configuration file of PostgreSQL server) with the calibrated values. In the following, we also report results based on calibrated cost models.

5.5.2 Results on the TPC-H Benchmark

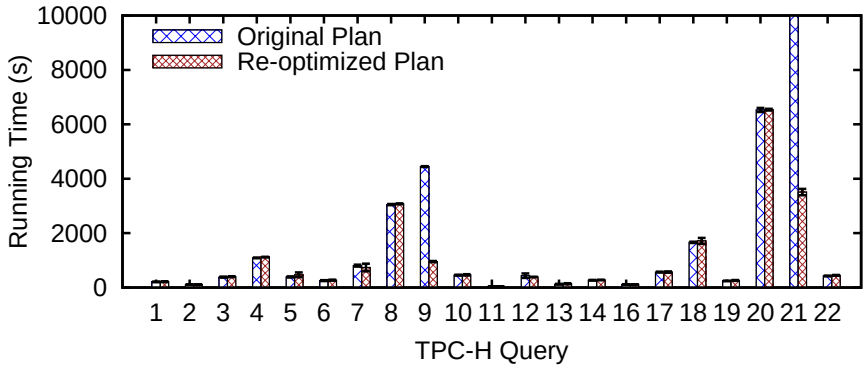
We tested 21 TPC-H queries. (We excluded Q15, which is not supported by our current implementation because it requires to create a view first.) For each TPC-H query, we randomly generated 10 instances. We cleared both the database buffer pool and the file system cache between each run of each query.

5.5.2.1 Results on Uniform Database

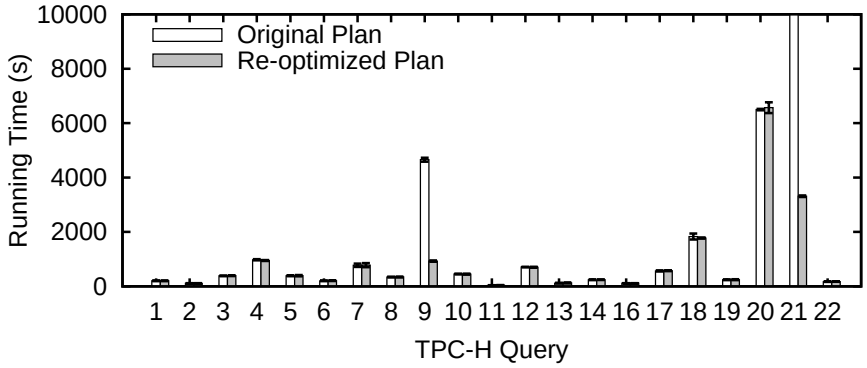
Figure 5.4 presents the average running times and their standard deviations (as error bars) of these queries over the uniform database.

We have two observations. First, while the running times for most queries almost do not change, we can see significant improvement for some queries. For example, as shown in Figure 5.4(a), even without calibration of the cost units, the average running time of Q9 drops from 4,446 seconds to only 932 seconds, a 79% reduction; more significantly, the average running time of Q21 drops from 20,746 seconds (i.e., almost 6 hours) to 3,508 seconds (i.e., less than 1 hour), a 83% reduction.

Second, calibration of the cost units can sometimes significantly reduce the running times for some queries. For example, comparing Figure 5.4(a) with Figure 5.4(b) we can observe that the average running time of Q8 drops from 3,048



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 5.4: Query running time over uniform 10GB TPC-H database ($z = 0$).

seconds to only 339 seconds, a 89% reduction, by just using calibrated cost units without even invoking the re-optimization procedure.

We further studied the re-optimization procedure itself. Figure 5.5 presents the number of plans generated during re-optimization. It substantiates our observation in Figure 5.4: for the queries whose running times were not improved, the re-optimization procedure indeed picked the same plans as those originally chosen by the optimizer. Figure 5.6 further compares the query running time excluding/including the time spent on re-optimization. For all the queries we tested, re-optimization time is ignorable compared to query execution time, which demonstrates the low overhead of our re-optimization procedure.

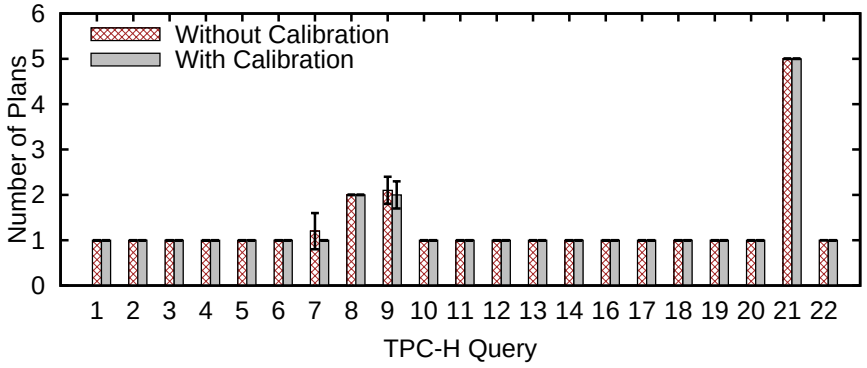
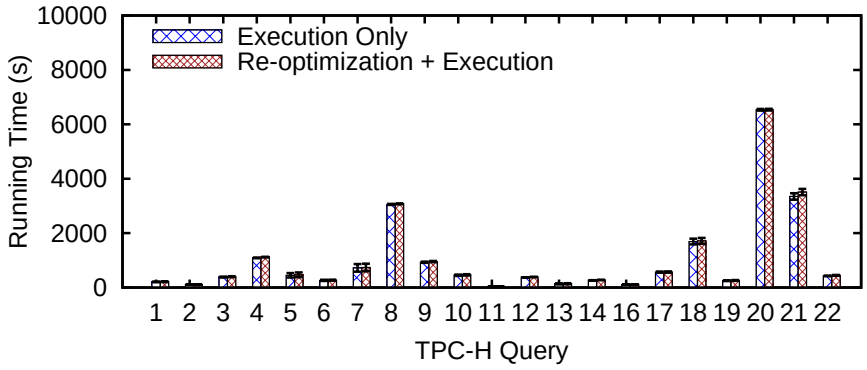


Figure 5.5: The number of plans generated during re-optimization over uniform 10GB TPC-H database ($z = 0$).

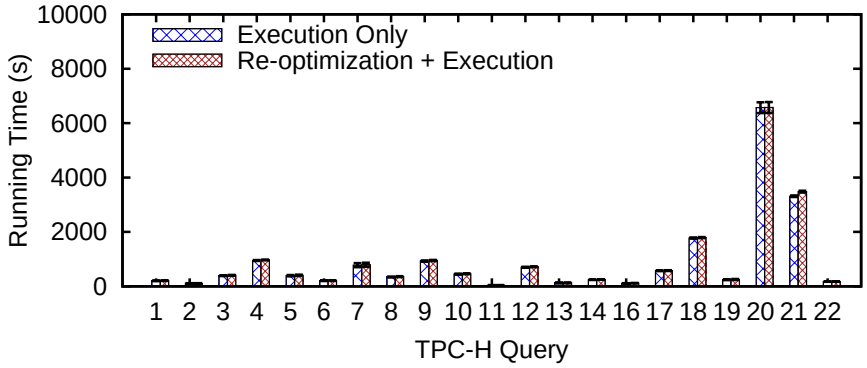
5.5.2.2 Results on Skewed Database

On the skewed database, we have observed results similar to that on the uniform database. Figure 5.7 presents the running times of the queries, with or without calibration of the cost units. While it looks quite similar to Figure 5.4, there is one interesting phenomenon not shown before. In Figure 5.7(a) we see that, without using calibrated cost units, the average running times for Q8 and Q9 actually increase after re-optimization. Recall that in Section 5.3.4 (specifically, Theorem 5.11) we have shown the local optimality of the plan returned by the re-optimization procedure. However, that result is based on the assumption that sampling-based cost estimates are consistent with actual costs (Assumption 2). Here this seems not the case. Nonetheless, after using calibrated cost units, both the running times of Q8 and Q9 were significantly improved (Figure 5.7(b)).

We further present the number of plans considered during re-optimization in Figure 5.8. Note that re-optimization seems to be more active on skewed data. Figure 5.9 shows the running times excluding/including the re-optimization times of the queries. Again, the additional overhead of re-optimization is trivial.



(a) Without calibration of the cost units

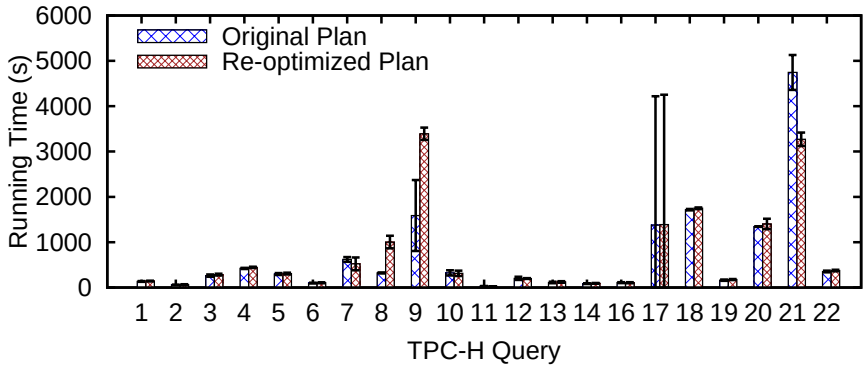


(b) With calibration of the cost units

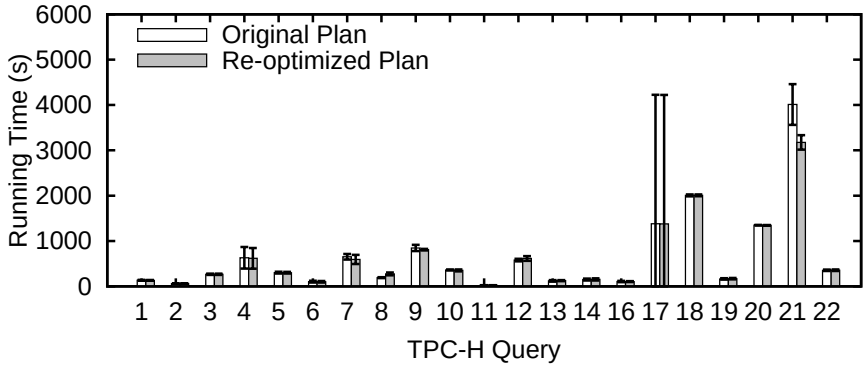
Figure 5.6: Query running time excluding/including re-optimization time over uniform 10GB TPC-H database ($z = 0$).

5.5.2.3 Discussion

While one might expect the chance for re-optimization to generate a better plan is higher on skewed databases, our experiments show that this may not be the case, at least for TPC-H queries. There are several different situations, though. First, if a query is too simple, then there is almost no chance for re-optimization. For example, Q1 contains no join, whereas Q16 and Q19 involve only one join so only local transformations are possible. Second, the final plan returned by the re-optimization procedure heavily relies on the initial plan picked by the optimizer, which is the seed or starting point where re-optimization originates. Note that,



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 5.7: Query running time over skewed 10GB TPC-H database ($z = 1$).

even if the optimizer has picked an inefficient plan, re-optimization cannot help if the estimated cost of that plan is not significantly erroneous. One question is if this is possible: the optimizer picks an inferior plan whose cost estimate is correct? This actually could happen because the optimizer may (incorrectly) overestimate the costs of the other plans in its search space. Another subtle point is that the inferior plan might be robust to certain degree of errors in cardinality estimates. Previous work has reported this phenomenon by noticing that the plan diagram (i.e., all possible plans and their governed optimality areas in the selectivity space) is dominated by just a couple of plans [77].

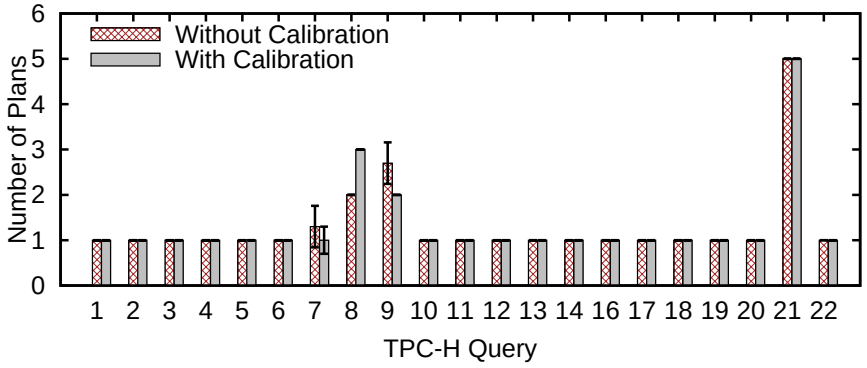


Figure 5.8: The number of plans generated during re-optimization over skewed 10GB TPC-H database ($z = 1$).

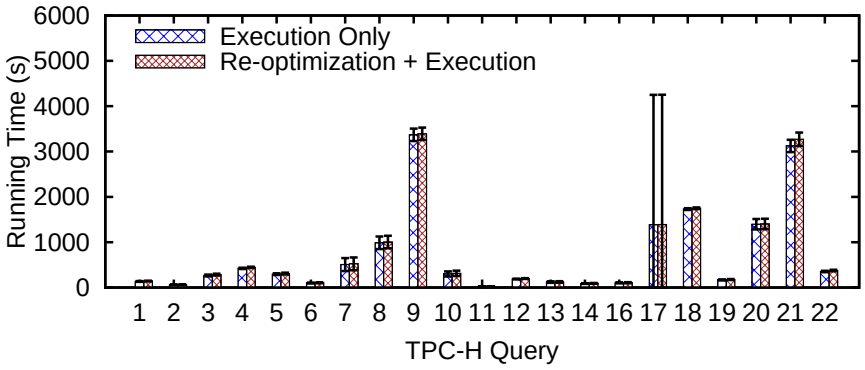
In summary, the effectiveness of re-optimization depends on factors that are out of the control of the re-optimization procedure itself. Nevertheless, we have observed intriguing improvement for some long-running queries by applying re-optimization, especially after calibration of the cost units.

5.5.3 Results of the Optimizer Torture Test

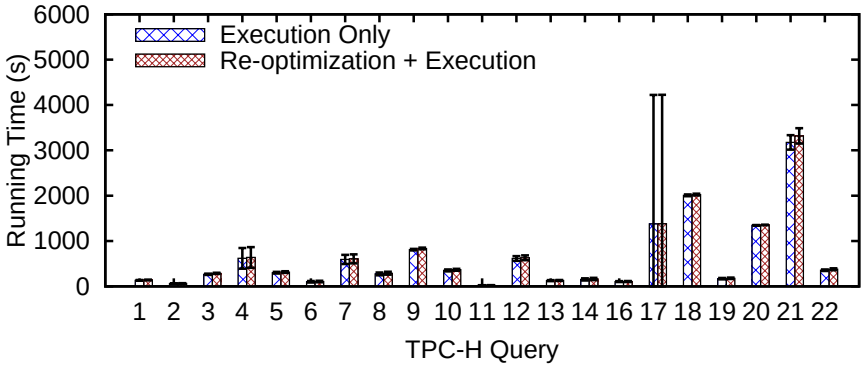
We created queries following our design of the OTT in Section 5.4.1. Specifically, if a query contains n tables (i.e., $n-1$ joins), we let m of the selections be $A = 0$ ($A = 1$), and let the remaining $n - m$ selections be $A = 1$ ($A = 0$). We generated two sets of queries: (1) $n = 5$ (4 joins), $m = 4$; and (2) $n = 6$ (5 joins), $m = 4$. Note that the maximal non-empty sub-queries then contain 3 joins over 4 tables with result size of roughly $100^4 = 10^8$ rows.[2] However, the size of each (whole) query is 0. So we would like to see the ability of the optimizer as well as the re-optimization procedure to identify the empty/non-empty sub-queries.

Figure 5.10 and 5.11 present the running times of the 4-join and 5-join queries, respectively. We generated in total 10 4-join queries and 30 5-join queries. Note that the y-axes are in log scale and we do not show queries that finish in less than 0.1

[2]Recall that a non-empty query must have equal A's (Equation (5.4)) and we generated data with roughly 100 rows per distinct value (Section 5.5.1).



(a) Without calibration of the cost units

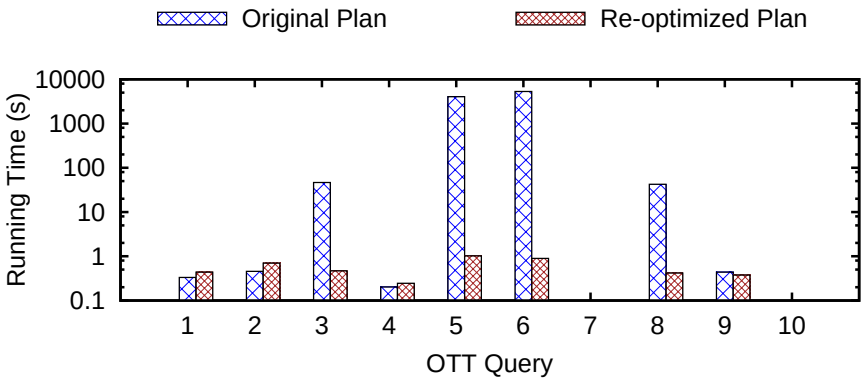


(b) With calibration of the cost units

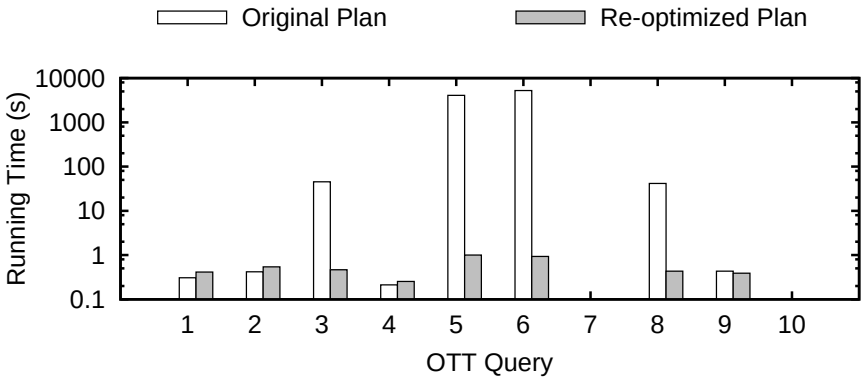
Figure 5.9: Query running time excluding/including re-optimization time over skewed 10GB TPC-H database ($z = 1$).

second. As we can see, sometimes the optimizer failed to detect the existence of empty sub-queries: it generated plans where empty join predicates were evaluated after the non-empty ones. The running times of these queries were then hundreds or even thousands of seconds. On the other hand, the re-optimization procedure did an almost perfect job in detecting empty joins, which led to very efficient query plans where the empty joins were evaluated first: all the queries after re-optimization finished in less than 1 second.

One might argue that the OTT queries are really contrived: they are hardly to see in real-world workloads. While this might be true, we think these queries serve our



(a) Without calibration of the cost units

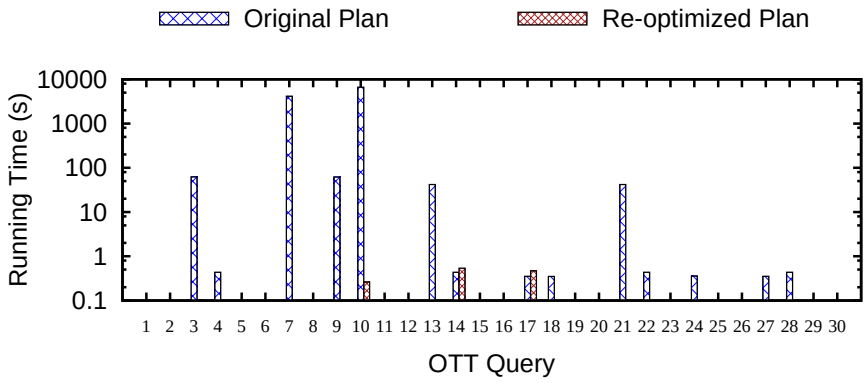


(b) With calibration of the cost units

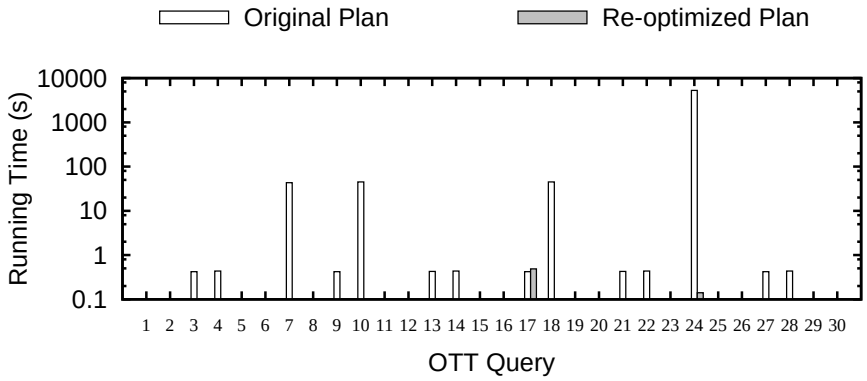
Figure 5.10: Query running times of 4-join queries.

purpose as exemplifying extremely hard cases for query optimization. Note that hard cases are not merely long-running queries: queries as simple as sequentially scanning huge tables are long-running too, but there is nothing query optimization can help with. Hard cases are queries where efficient execution plans do exist but it might be difficult for the optimizer to find them. The OTT queries are just these instances. Based on the experimental results of the OTT queries, re-optimization is helpful to give the optimizer second chances if it initially made a bad decision.

Another concern is if commercial database systems could do a better job on the OTT queries. In regard of this, we also ran the OTT over two major commercial database systems. The performance is very similar to that of PostgreSQL (Figure 5.12



(a) Without calibration of the cost units



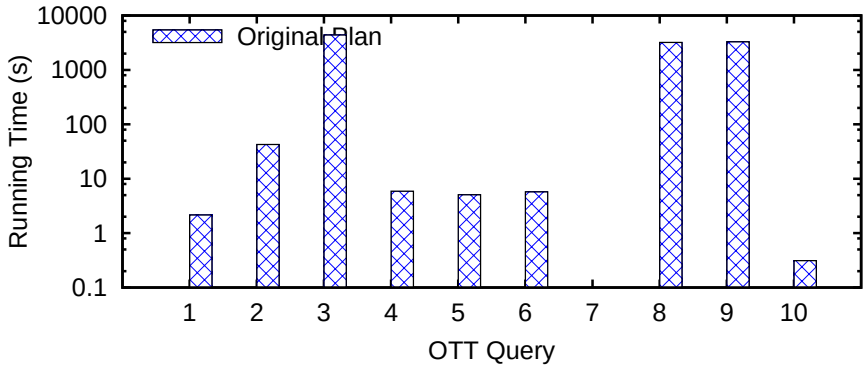
(b) With calibration of the cost units

Figure 5.11: Query running times of 5-join queries.

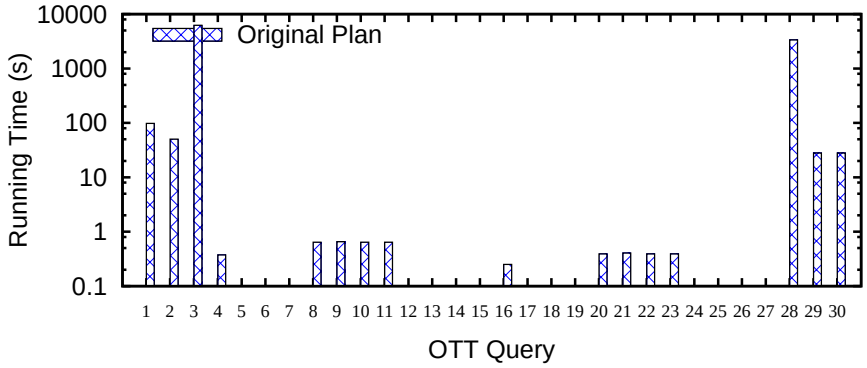
and 5.13). We therefore speculate that commercial systems could also benefit from our re-optimization technique proposed in this chapter.

5.5.4 A Note on Multidimensional Histograms

Note that even using multidimensional histograms (e.g., [18, 68, 73]) may not be able to detect the data correlation presented in the OTT queries, unless the buckets are so fine-grained that the exact joint distributions are retained. To understand this, let us consider the following example.



(a) 4-join OTT queries

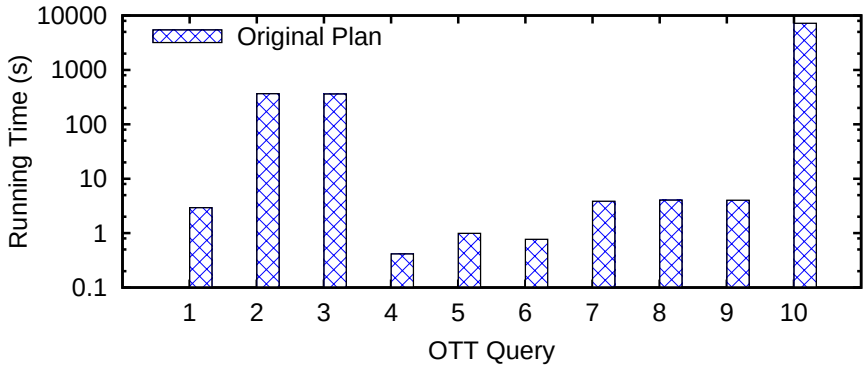


(b) 5-join OTT queries

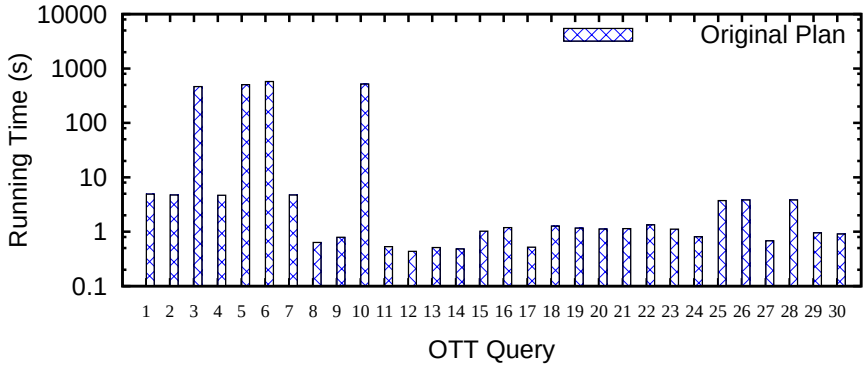
Figure 5.12: Query running times of the OTT queries on the system A.

Example 5.18. *Following our design of OTT, suppose that now we only have two tables* $R_1(A_1, B_1)$ *and* $R_2(A_2, B_2)$*. Moreover, suppose that each* A_k *(and thus* B_k*) contains* $m = 2l$ *distinct values, and we construct (perfect) 2-dimensional histograms on* (A_k, B_k) *(*$k = 1, 2$*). Each dimension is evenly divided into* $\frac{m}{2} = l$ *intervals, so each histogram contains* l^2 *buckets. The joint distribution over* (A_k, B_k) *estimated by using the histogram is then:*

$$\begin{cases} \Pr(2r-2 \leqslant A_k < 2r, 2r-2 \leqslant B_k < 2r) = \frac{1}{l}, & 1 \leqslant r \leqslant l; \\ \Pr(a_l \leqslant A_k < a_2, b_1 \leqslant B_k < b_2) = 0, & \textit{other buckets.} \end{cases}$$



(a) 4-join OTT queries



(b) 5-join OTT queries

Figure 5.13: Query running times of the OTT queries on the system B.

For instance, if $m = 100$*, then* $l = 50$*. So we have*

$$\Pr(0 \leqslant A_k < 2, 0 \leqslant B_k < 2) = \cdots = \Pr(98 \leqslant A_k < 100, 98 \leqslant B_k < 100) = \frac{1}{50},$$

while all the other buckets are empty. On the other hand, the actual joint distribution is

$$\Pr(A_k = a, B_k = b) = \begin{cases} \frac{1}{m}, & \text{if } a = b; \\ 0, & \text{otherwise.} \end{cases}$$

Now, let us consider the selectivities for two OTT queries:

(q_1) $\sigma_{A_1=0}(R_1) \bowtie \sigma_{A_2=1}(R_2) = \sigma_{A_1=0 \wedge A_2=1 \wedge B_1=B_2}(R_1 \times R_2)$;

(q_2) $\sigma_{A_1=0}(R_1) \bowtie \sigma_{A_2=0}(R_2) = \sigma_{A_1=0 \wedge A_2=0 \wedge B_1=B_2}(R_2 \times R_2)$.

We know that q_1 *is empty but* q_2 *is not. However, the estimated selectivity (and thus cardinality) of* q_1 *and* q_2 *is the same by using the 2-dimensional histogram. To see this, let us compute the estimated selectivity* $\hat{s}_1$ *of* q_1*. We have*

$$\begin{aligned}
\hat{s}_1 &= \Pr(A_1 = 0, A_2 = 1, B_1 = B_2) \\
&= \sum\nolimits_{0 \leqslant c \leqslant m-1} \Pr(A_1 = 0, A_2 = 1, B_1 = c, B_2 = c) \\
&= \sum\nolimits_{c=0,1} \Pr(A_1 = 0, A_2 = 1, B_1 = c, B_2 = c) \\
&= \sum\nolimits_{c=0,1} \Pr(A_1 = 0, B_1 = c) \times \Pr(A_2 = 1, B_2 = c) \\
&= \sum\nolimits_{c=0,1} (\frac{1}{l} \cdot \frac{1}{4}) \times (\frac{1}{l} \cdot \frac{1}{4}) = \frac{1}{8l^2}.
\end{aligned}$$

The last step follows from the assumption used by histograms that data inside each bucket is uniformly distributed. We can conclude that the estimated selectivity of q_2 *is* $\hat{s}_2 = \hat{s}_1 = \frac{1}{8l^2}$*, by simply replacing* $A_2 = 1$ *with* $A_2 = 0$ *in the above derivation. With the setting used in our experiments,* $m = 100$ *and thus* $l = 50$*. So each 2-dimensional histogram contains* $l^2 = 2,500$ *buckets. However, even such detailed histograms cannot help the optimizer distinguish empty joins from nonempty ones. Furthermore, note that our conclusion is independent of* m*, while the number of buckets increases quadratically in* m*. For instance, when* $m = 10^4$ *which means we have histograms containing* 2.5×10^7 *buckets, the optimizer still cannot rely on the histograms to generate efficient query plans for OTT queries.*

5.6 Related Work

Query optimization has been studied for decades, and we refer the readers to [20] and [52] for surveys in this area.

Cardinality estimation is a critical problem in cost-based query optimization, and has triggered extensive research in the database community. Approaches for cardinality estimation in the literature are either static or dynamic. Static approaches

usually rely on various statistics that are collected and maintained periodically in an off-line manner, such as histograms (e.g., [53, 73]), samples (e.g., [19, 44, 62]), sketches (e.g., [12, 80]), or even graphical models (e.g. [37, 89]). In practice, approaches based on histograms are dominant in the implementations of current query optimizers. However, histogram-based approaches have to rely on the notorious attribute-value-independence (AVI) assumption, and they often fail to capture data correlations, which could result in significant errors in cardinality estimates. While variants of histograms (in particular, multidimensional histograms, e.g., [18, 68, 73]) have been proposed to overcome the AVI assumption, they suffer from significantly increased overhead on large databases. Meanwhile, even if we can afford the overhead of using multidimensional histograms, they are still insufficient in many cases, as we discussed in Section 5.5.4. Compared with histogram-based approaches, sampling is better at capturing data correlation. One reason for this is that sampling evaluates queries on real rather than summarized data.

On the other hand, dynamic approaches further utilize information gathered during query runtime. Approaches in this direction include dynamic query plans (e.g., [29, 40]), parametric query optimization (e.g. [54]), query feedback (e.g., [17, 85]), mid-query re-optimization (e.g. [56, 66]), and quite recently, plan bouquets [34]. The ideas behind dynamic query plans and parametric query optimization are similar: rather than picking one single optimal query plan, all possible optimal plans are retained and the decision on which one to use is deferred until runtime. Both approaches suffer from the problem of combinatorial explosion and are usually used in contexts where expensive pre-compilation stages are affordable. The recent development of plan bouquets [34] is built on top of parametric query optimization so it may also incur a heavy query compilation stage. Unlike standard parametric query optimization that sticks to a single plan at runtime, it chooses a pool of candidate plans and runs them one by one with respect to the increasing order of their estimated costs. While this idea is effective in quickly identifying bad plans that are due to underestimation of their costs, it would increase query running time in general. Therefore, this idea seems to be more attractive if robustness is more important than average performance.

Meanwhile, approaches based on query feedback record statistics of past queries and use this information to improve cardinality estimates for future queries. Some of these approaches have been adopted in commercial systems such as IBM DB2 [85] and Microsoft SQL Server [17]. Nonetheless, collecting query feedback incurs additional runtime overhead as well as storage overhead of ever-growing volume of statistics. Moreover, the effectiveness of using a mixture of actual/estimated statistics for parts of plans that have/have not been observed in past queries remains debatable. That is, it is unclear if the optimizer could bring up a better plan by giving it partially improved cardinalities [25]. While our proposed framework faces the same issue, we remedy that by introducing the sampling-based validation stage.

The most relevant work in the literature is perhaps the line along mid-query re-optimization [56, 66]. We have articulated the connection between our work and this work in Section 5.1. In some sense, our approach sits between static and dynamic approaches. We combine the advantage of lower overheads from static approaches and the advantage of more optimization opportunities from dynamic approaches. This compromise leads to a lightweight query re-optimization procedure that could bring up better query plans.

Finally, we note that we are not the first that investigates the idea of incorporating sampling into query optimization. Ilyas et al. proposed using sampling to detect data correlations and then collecting joint statistics for those correlated data columns [51]. However, this seems to be insufficient if data correlation is caused by specific selection predicates, such as those OTT queries used in our experiments. Babcock and Chaudhuri also investigated the usage of sampling in developing a robust query optimizer [14]. While robustness is another interesting goal for query optimizer, it is beyond the scope of this chapter.

5.7 Summary

In this chapter, we studied the problem of incorporating sampling-based cardinality estimates into query optimization. We proposed an iterative query re-optimization procedure that supplements the optimizer with refreshed cardinality estimates via

sampling and gives it second chances to generate better query plans. We show the efficiency and effectiveness of this re-optimization procedure both theoretically and experimentally. While we have demonstrated that our approach can catch and fix some bad plans, we are not saying this is the only way to do that — in particular, in some cases creating more comprehensive histograms (e.g., multidimensional histograms) would also help. However, it may be infeasible to cover all cases. For example, we may need too many histograms on large databases with complex schemas and therefore cannot afford the overhead. Multidimensional histograms may also fail on cases that can be handled by using sampling, as shown in Section 5.5.4. By using sampling as a post-processing step, it is actually orthogonal to the types of histograms used by the optimizer: the optimizer is free to use multidimensional histograms to improve the query plan it returns, which can be further improved by using our sampling-based re-optimization procedure.

There are several directions worth further exploring. First, as we have mentioned, the initial plan picked by the optimizer would have great impact on the final plan returned by re-optimization. While it remains interesting to study this impact theoretically, it might also be an intriguing idea to think about varying the way that the query optimizer works. For example, rather than just returning one plan, the optimizer could return several candidates and let the re-optimization procedure work on each of them. This might make up for the potentially bad situation currently faced by the re-optimization procedure that it may start with a bad seed plan. Second, the re-optimization procedure itself could be further improved. As an example, note that in this chapter we let the optimizer unconditionally accept cardinality estimates by sampling. However, sampling is by no means perfect. A more conservative approach is to consider the uncertainty of the cardinality estimates returned by sampling as well. In Chapter 4, we investigated the problem of quantifying uncertainty in sampling-based query running time estimation. It is very interesting to study a combination of that framework with the query re-optimization procedure proposed in this chapter. We leave all these as promising areas for future work.

Chapter 6

Conclusion

In this dissertation, we developed sampling-based techniques that are useful for query execution time prediction and query optimization. Our key idea is to use sampling as a post-processing, validation step that is only applied to the final plan returned by the query optimizer. We show that, with little additional overhead, this idea can help refine cardinality estimates and therefore query execution time predictions as well. It can further help quantify the uncertainty in cardinality estimates and thus query execution time predictions. Moreover, by introducing a feedback loop from sampling to the query optimizer, we can improve the final query plan returned by the optimizer via an iterative re-optimization procedure. Figure 6.1 highlights this high-level big picture.

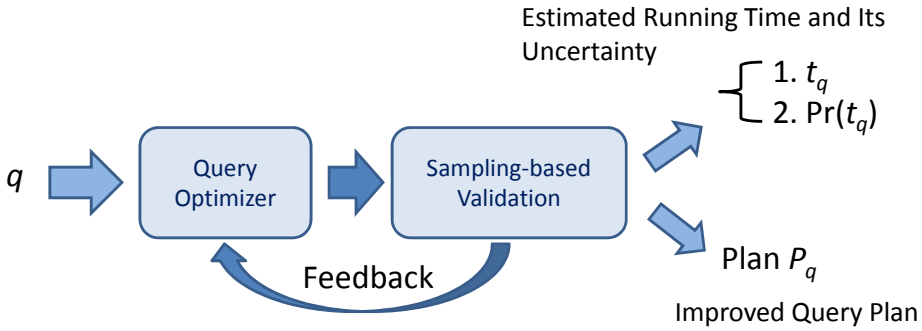


Figure 6.1: Use sampling as a post-processing, validation step.

Although we have discussed interesting directions for future work in each chapter, we have some further thoughts regarding the idea of using sampling-based techniques in query optimization. While sampling-based cardinality estimation techniques have been extensively studied in the literature, they are rarely used in current query optimizer implementations, mainly due to the runtime overhead they would incur. It is perhaps time to rethink the role of sampling, given the ever-growing memory sizes of modern computers. As demonstrated in this dissertation, the additional overhead of sampling is usually ignorable on large databases. If samples could be memory-resident, we would expect a further drop in this overhead, which, as a result, would allow us to use sampling-based techniques to validate more query plans. At a higher level, there is a trade-off between the time that could be spent on query optimization and the quality of the final query plan. Given that cardinality estimates are crucial for the accuracy of cost estimates and sampling can help improve cardinality estimates, we believe that it is worth to use sampling more intensively if its overhead is manageable. One possible way, as we have discussed in Chapter 5, is to bring uncertainty into query optimization and use sampling for those query plans about which traditional, histogram-based approaches have little confidence. Of course, it remains an interesting question to study how to measure the uncertainty or confidence for cost estimates based on histograms. We hope our work in Chapter 5 might encourage further research in this area.

Finally, we would like to give some critiques to the techniques developed in this dissertation. First, as we mentioned, the approach we used to calibrate cost units in Chapter 2 is based on the linearity of PostgreSQL's cost model. While cost models of other database systems are similar, they, especially those of commercial database systems, might be more complicated and involve more parameters. We therefore view our work only as a case study that points out basic elements in cost models and general principles of calibration.

Second, while the sampling-based selectivity estimator we used performs well in practice, we should note here that there was also some theoretic negative result on random sampling over joins [22]. It suggests that, in certain cases, it is impossible to take a random sample from a join by using uniform samples over base tables,

unless the sample sizes are comparable to the base table sizes. However, this could only happen when the join is highly selective. In this case, the estimated selectivity would be 0 unless the sample sizes are sufficiently large. In fact, if we keep sampling (rather than taking a fixed amount of samples), we will observe large sample variances. Nonetheless, the problem of selectivity/cardinality estimation is different from that of taking random samples from join results. When the join selectivity is very small, an estimate of 0 is perhaps still useful. This only becomes an issue if the join result needs to be further joined with another huge table that might lead to nontrivial join size. Another problem that is not discussed in this dissertation is how to update samples when the underlying database changes. While periodically updating samples is one option, there was some interesting previous work exploring the idea of utilizing execution feedback [58]: samples are updated if the estimates turn out to be far away from the actual cardinalities.

Third, cost estimation is not just cardinality estimation; its accuracy further depends on the cost model itself. Our calibration techniques in Chapter 2 and 3 assumed that the cost models used by current optimizers are reasonably accurate. However, this is not the case for some queries, and we still observe significant errors over these queries. Our work on measuring uncertainty in query execution time estimation in Chapter 4 also did not address the inaccuracy in cost models. A study in this direction is desirable but seems quite challenging. Cost models are currently handcrafted by query optimizer developers, and there is little work in the literature on systematic, formal approaches for cost modeling. Partially due to the difficulty in formalizing cost models, our study in Chapter 5 on re-optimizing query by using sampling-based cardinality estimates still lacks theoretical guarantees on the optimality of the final query plan. In the bigger picture, the relationship between the accuracy of cost modeling and the optimality of query plan returned by the optimizer remains unclear. So far, we are not aware of any theoretical work on this important problem. We would like to treat all these critiques as interesting, unsolved questions that call for future research.

Appendix A

Theoretic Results

In this appendix, we present theoretic results (e.g., proofs of certain lemmas and theorems) that are too long to be included in the main text of this dissertation.

A.1 Proof of Lemma 2.8

Proof. The proof idea is briefly sketched in Appendix D of [44] but not fully developed. For completeness purpose we provide a proof here following that sketch.

Define $X(i,j) = 1$ if the block we take from the relation R_1 in the i-th sampling step is $B(1,j)$, and $X(i,j) = 0$ otherwise, where $1 \leqslant j \leqslant m_1$. Define $\mathbf{X}_i = (X(i,1), ..., X(i,m_1))$. Since we sample with replacement, $\mathbf{X}_1$, ..., $\mathbf{X}_n$ are i.i.d. random vectors with (common) expectation

$$\mu_X = E[\mathbf{X}_i] = (\frac{1}{m_1}, ..., \frac{1}{m_1}).$$

Let

$$\bar{\mathbf{X}}_n = \frac{1}{n}\sum_{i=1}^{n} \mathbf{X}_i = (\bar{X}(n,1), ..., \bar{X}(n,m_1)),$$

where $\bar{X}(n,j) = \frac{1}{n}\sum_{i=1}^{n} X(i,j)$. Then by the strong law of large numbers,

$$\Pr\Big[\lim_{n\to\infty} \bar{\mathbf{X}}_n = \mu_{\mathbf{X}}\Big] = 1.$$

On the other hand, define

$$Y(n,j) = \frac{1}{n^{K-1}} \sum_{i_2=1}^{n} \cdots \sum_{i_K=1}^{n} \rho_{\mathbf{B}(j,L_{2,i_2},\ldots,L_{K,i_K})},$$

and $\mathbf{Y}_n = (Y(n,1),\ldots,Y(n,m_1))$. We have

$$\mu_{\mathbf{Y}} = \mathrm{E}\big[\mathbf{Y}_n\big] = (\rho_1,\rho_2,\ldots,\rho_{m_1}),$$

where

$$\rho_j = \frac{1}{\prod_{k=2}^{K} m_k} \sum_{i_2=1}^{m_2} \cdots \sum_{i_K=1}^{m_K} \rho_{\mathbf{B}(j,i_2,\ldots,i_K)}.$$

Note that we can now write Equation (2.2) as

$$\tilde{\rho}_{\mathbf{R}}^{cp} = \bar{\mathbf{X}}_n \cdot \mathbf{Y}_n = \sum_{j=1}^{m_1} \bar{X}(n,j) Y(n,j).$$

Our next goal is to show that $\Pr\big[\lim_{n\to\infty} \mathbf{Y}_n = \mu_{\mathbf{Y}}\big] = 1$. If this is true, then we have

$$\Pr\Big[\lim_{n\to\infty} \tilde{\rho}_{\mathbf{R}}^{cp} = \mu_{\mathbf{X}} \cdot \mu_{\mathbf{Y}}\Big] = 1.$$

Since

$$\begin{aligned}
\mu_{\mathbf{X}} \cdot \mu_{\mathbf{Y}} &= \frac{1}{m_1} \sum_{j=1}^{m_1} \frac{1}{\prod_{k=2}^{K} m_k} \sum_{i_2=1}^{m_2} \cdots \sum_{i_K=1}^{m_K} \rho_{\mathbf{B}(j,i_2,\ldots,i_K)} \\
&= \frac{1}{\prod_{k=1}^{K} m_k} \sum_{i_1=1}^{m_1} \cdots \sum_{i_K=1}^{m_K} \rho_{\mathbf{B}(i_1,i_2,\ldots,i_K)} = \rho_{\mathbf{R}},
\end{aligned}$$

we therefore prove the strong consistency of $\tilde{\rho}_{\mathbf{R}}^{cp}$.

We now show that $\Pr\big[\lim_{n\to\infty} \mathbf{Y}_n = \mu_{\mathbf{Y}}\big] = 1$ by induction on K. When $K = 2$,

$$Y(n, j) = \frac{1}{n} \sum_{i_2=1}^{n} \rho_{\mathbf{B}(j, L_{2,i_2})}.$$

Note that, with a fixed j, the $\rho_{\mathbf{B}(j,L_{2,i_2})}$'s are i.i.d. random variables. So by the strong law of large numbers, $\Pr\big[\lim_{n\to\infty} Y(n, j) = \rho_j\big] = 1$, and hence $\Pr\big[\lim_{n\to\infty} \mathbf{Y}_n = \mu_{\mathbf{Y}}\big] = 1$. Now assume that the strong consistency holds for K relations. Consider the case of $K + 1$ relations:

$$Y(n, j) = \frac{1}{n^K} \sum_{i_2=1}^{n} \cdots \sum_{i_{K+1}=1}^{n} \rho_{\mathbf{B}(j, L_{2,i_2}, \ldots, L_{K+1,i_{K+1}})}.$$

Define

$$Y(n, j, i_2) = \frac{1}{n^{K-1}} \sum_{i_3=1}^{n} \cdots \sum_{i_{K+1}=1}^{n} \rho_{\mathbf{B}(j, L_{2,i_2}, \ldots, L_{K+1,i_{K+1}})}.$$

By induction hypothesis,

$$\Pr\big[\lim_{n\to\infty} Y(n, j, i_2) = \rho_{j,i_2}\big] = 1,$$

where

$$\rho_{j,i_2} = \frac{1}{\prod_{k=3}^{K+1} m_k} \sum_{i_3=1}^{m_2} \cdots \sum_{i_{K+1}=1}^{m_{K+1}} \rho_{\mathbf{B}(j, i_2, \ldots, i_{K+1})}.$$

Since

$$Y(n, j) = \frac{1}{n} \sum_{i_2=1}^{n} Y(n, j, i_2),$$

and for fixed j, $Y(n, j, i_2)$ are i.i.d. random variables with (common) mean

$$E\big[Y(n, j, i_2)\big] = \frac{1}{m_2} \sum_{i_2=1}^{m_2} \rho_{j,i_2} = \frac{1}{\prod_{k=2}^{K+1} m_k} \sum_{i_2=1}^{m_2} \cdots \sum_{i_{K+1}=1}^{m_{K+1}} \rho_{\mathbf{B}(j, i_2, \ldots, i_{K+1})} = \rho_j.$$

We hence have

$$\Pr\Big[\lim_{n\to\infty} Y(n,j) = \rho_j\Big] = 1,$$

by applying the strong law of large numbers again. This completes the proof. □

A.2 Variance of The Selectivity Estimator

The variance of the selectivity estimator ρ_n (ref. Section 4.5), unfortunately, is nontrivial when writing it mathematically:

Theorem A.1. *The variance of ρ_n is [44]:*

$$\begin{aligned} \mathrm{Var}[\rho_n] &= \sum_{r=1}^{K} \frac{(n-1)^{K-r}}{n^K} \\ &\times \sum_{S\in\mathcal{S}_r} \left(\frac{1}{|\Lambda(S)|} \sum_{\mathbf{l}\in\Lambda(S)} (\rho_S(\mathbf{l}) - \rho)^2 \right). \end{aligned} \tag{A.1}$$

Here $S = \{k_1, ..., k_r\} \subseteq \{1, 2, ..., K\}$ such that $k_1 < k_2 < \cdots < k_r$, $\mathcal{S}_r$ is the collection of all subsets of $\{1, 2, ..., K\}$ with size r (for $1 \leqslant r \leqslant K$), and $\Lambda(S)$ is defined to be

$$\Lambda(S) = \{1, 2, ..., m_{k_1}\} \times \cdots \times \{1, 2, ..., m_{k_r}\}.$$

Moreover, for $\mathbf{l} = (l_1, ..., l_r) \in \Lambda(S)$, $\rho_S(\mathbf{l})$ is the average selectivity over $\mathbf{B}(L_1, ..., L_K)$ such that $L_{k_j} = l_j$ $(1 \leqslant j \leqslant r)$. For example, if $K = 4$, $S = \{2, 3\}$ and $\mathbf{l} = (8, 9)$, then

$$\rho_S(\mathbf{l}) = (m_1 m_4)^{-1} \sum_{L_1=1}^{m_1} \sum_{L_4=1}^{m_4} \rho_{\mathbf{B}(L_1,8,9,L_4)}.$$

We next prove Theorem A.1. Roughly speaking, the idea of the proof is to first partition the samples into groups based on how many blocks they share, then compute the variance of each group, and finally sum them up. We start with the following standard result from probability theory:

Lemma A.2. *Let* $X_1, ..., X_n$ *be* n *random variables, then*

$$\mathrm{Var}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{Cov}(X_i, X_j),$$

where $\mathrm{Cov}(X_i, X_j)$ *is the covariance of* X_i *and* X_j*:*

$$\mathrm{Cov}(X_i, X_j) = \mathrm{E}[(X_i - \mathrm{E}[X_i])(X_j - \mathrm{E}[X_j])].$$

Now define $\mathcal{X}$ to be the set of all sample blocks, namely,

$$\mathcal{X} = \{\rho_{\mathbf{B}(L_{1,i_1},\cdots,L_{K,i_K})} | 1 \leqslant i_k \leqslant n, 1 \leqslant k \leqslant K\}.$$

Based on Lemma A.2 and Equation (4.4), we have

$$\mathrm{Var}[\rho_n] = \frac{1}{(n^K)^2} \sum_{X\in\mathcal{X}} \sum_{X'\in\mathcal{X}} \mathrm{Cov}(X, X'). \tag{A.2}$$

Consider any $X = \rho_{\mathbf{B}(j_1,...,j_K)}$ and $X' = \rho_{\mathbf{B}(j'_1,...,j'_K)}$ in the summands of Equation (A.2). If $j_k \neq j'_k$ for $1 \leqslant k \leqslant K$, then X and X' are independent and $\mathrm{Cov}(X, X') = 0$. Hence, only X and X' that share at least one common coordinate will contribute a non-zero summand to Equation (A.2). We thereby partition the pairs (X, X') according to the number of coordinates they share. Specifically, for $S = \{k_1, ..., k_r\} \subseteq \{1, 2, ..., K\}$, we denote $X \sim_S X'$ if $j_{k_m} = j'_{k_m}$ for $1 \leqslant m \leqslant r$. This gives us the following equivalent expression for $\mathrm{Var}[\rho_n]$:

$$\mathrm{Var}[\rho_n] = \frac{1}{(n^K)^2} \sum_{r=1}^{K} \sum_{S\in\mathcal{S}_r} \sum_{X\sim_S X'} \mathrm{Cov}(X, X'). \tag{A.3}$$

Lemma A.3. *For a fixed* $S \in \mathcal{S}_r$*, the number of pairs* (X, X') *such that* $X \sim_S X'$ *is* $(n-1)^{K-r}n^K$*. As a result,* $\mathrm{Var}[\rho_n]$ *can be further expressed as:*

$$\mathrm{Var}[\rho_n] = \sum_{r=1}^{K} \frac{(n-1)^{K-r}}{n^K} \times \sum_{S \in \mathcal{S}_r} \mathrm{Cov}(X, X'). \tag{A.4}$$

Our next goal is to give an expression for $\mathrm{Cov}(X, X')$ when $X \sim_S X'$, as shown in Lemma A.4. Equation (A.1) in Theorem A.1 then follows by combining Lemma A.3 and A.4. This completes the proof of Theorem A.1.

Lemma A.4. *If* $X \sim_S X'$*, then*

$$\mathrm{Cov}(X, X') = \frac{1}{|\Lambda(S)|} \sum_{\mathbf{l} \in \Lambda(S)} (\rho_S(\mathbf{l}) - \rho)^2.$$

Proof. We have

$$\mathrm{Cov}(X, X') = \mathrm{E}_{X \sim_S X'}[(X - \mathrm{E}[X])(X' - \mathrm{E}[X'])].$$

Since $\mathrm{E}[X] = \mathrm{E}[X'] = \rho$, it follows that

$$\mathrm{Cov}(X, X') = \mathrm{E}_{X \sim_S X'}[(X - \rho)(X' - \rho)].$$

We further denote $X \sim_{S(\mathbf{l})} X'$ for $\mathbf{l} = (l_1, ..., l_r)$, if $X \sim_S X'$ and $j_{k_m} = l_m$ for $1 \leqslant m \leqslant r$. We then have

$$\mathrm{Cov}(X, X') = \frac{1}{|\Lambda(S)|} \sum_{\mathbf{l} \in \Lambda(S)} \mathrm{E}_{X \sim_{S(\mathbf{l})} X'}[(X - \rho)(X' - \rho)].$$

Now consider $\mathrm{E}_{X \sim_{S(\mathbf{l})} X'}[(X - \rho)(X' - \rho)]$. By definition, it is the average of the following quantities

$$\gamma = (\rho_{\mathbf{B}(j_1, ..., j_K)} - \rho)(\rho_{\mathbf{B}(j'_1, ..., j'_K)} - \rho)$$

by setting $j_{k_m} = j'_{k_m} = l_m$ for $1 \leqslant m \leqslant r$. Let $S^c = \{1, ..., K\} - S$ be the complement of S. We then have

$$E = E_{X \sim_{S(l)} X'}[(X - \rho)(X' - \rho)] = \big(\frac{1}{|\Lambda(S^c)|}\big)^2 \sum_{\Lambda(S^c)} \sum_{\Lambda(S^c)} \gamma.$$

After some rearrangement of the summands, we can have

$$\begin{aligned} E &= \big(\frac{1}{|\Lambda(S^c)|}\big)^2 \big(\sum_{\Lambda(S^c)} (\rho_{\mathbf{B}(j_1,...,j_K)} - \rho)\big)^2 \\ &= \big(\frac{1}{|\Lambda(S^c)|} \sum_{\Lambda(S^c)} (\rho_{\mathbf{B}(j_1,...,j_K)} - \rho)\big)^2 \\ &= \big((\frac{1}{|\Lambda(S^c)|} \sum_{\Lambda(S^c)} \rho_{\mathbf{B}(j_1,...,j_K)}) - \rho\big)^2 \\ &= (\rho_S(\mathbf{l}) - \rho)^2. \end{aligned}$$

This completes the proof of the lemma. □

A.3 A Tighter Upper Bound for Covariance

Consider two operators O and O' where $O \in Desc(O')$. Suppose that $|\mathcal{R}| = K$, $|\mathcal{R}'| = K'$, and $|\mathcal{R} \cap \mathcal{R}'| = m$ ($m \geqslant 1$). Let the estimators for O and O' be ρ_n and ρ'_n where n is the number of sample steps, and define $\rho = E[\rho_n]$ and $\rho' = E[\rho'_n]$.

Theorem A.5. *Let $\mathcal{S}_r$, $\Lambda(S)$, and $\rho_S(\mathbf{l})$ be the same as that defined in Theorem A.1. Define*

$$\sigma_S^2 = \frac{1}{|\Lambda(S)|} \sum_{\mathbf{l} \in \Lambda(S)} (\rho_S(\mathbf{l}) - \rho)^2,$$

and

$$S_\rho^2(m, n) = \sum_{r=1}^{m} \big(1 - \frac{1}{n}\big)^{m-r} \big(\frac{1}{n}\big)^r \sum_{S \in \mathcal{S}_r} \sigma_S^2.$$

We then have

$$|\operatorname{Cov}(\rho_n, \rho'_n)| \leqslant \sqrt{S^2_\rho(m,n) S^2_{\rho'}(m,n)} \leqslant \sqrt{\operatorname{Var}[\rho_n] \operatorname{Var}[\rho'_n]}.$$

We next prove Theorem A.5. To establish the first inequality, namely,

$$|\operatorname{Cov}(\rho_n, \rho'_n)| \leqslant \sqrt{S^2_\rho(m,n) S^2_{\rho'}(m,n)},$$

we need two lemmas. The first one gives an explicit expression of the covariance $\operatorname{Cov}(\rho_n, \rho'_n)$, which is similar to the expression of $\operatorname{Var}[\rho_n]$ shown in Theorem A.1.

Lemma A.6. *Let $\mathcal{S}_r$ be the collection of all subsets of $\mathcal{R}$ with size r (for $1 \leqslant r \leqslant m$). Define*

$$\operatorname{Cov}_S(\rho, \rho') = \frac{1}{|\Lambda(S)|} \sum_{\mathbf{l} \in \Lambda(S)} (\rho_S(\mathbf{l}) - \rho)(\rho'_S(\mathbf{l}) - \rho').$$

Then

$$\operatorname{Cov}(\rho_n, \rho'_n) = \sum_{r=1}^{m} \frac{(n-1)^{m-r}}{n^m} \sum_{S \in \mathcal{S}_r} \operatorname{Cov}_S(\rho, \rho').$$

Here $\rho_S(\mathbf{l})$ and $\rho'_S(\mathbf{l})$ are the same as that in Theorem A.1, defined over $\mathcal{R}$ and $\mathcal{R}'$.

Proof. The idea is similar to our proof of Theorem A.1. Let $K = |\mathcal{R}|$ and $K' = |\mathcal{R}'|$. We have

$$\rho_n = \frac{1}{n^K} \sum_{i_1=1}^{n} \cdots \sum_{i_K=1}^{n} \rho_{\mathbf{B}(L_{1,i_1}, \cdots, L_{K,i_K})},$$

and

$$\rho'_n = \frac{1}{n^{K'}} \sum_{i_1=1}^{n} \cdots \sum_{i_{K'}=1}^{n} \rho'_{\mathbf{B}(L_{1,i_1}, \cdots, L_{K,i_{K'}})}.$$

Therefore,

$$\operatorname{E}[\rho_n \rho'_n] = \frac{1}{n^{K+K'}} \sum_{k=1}^{n^K} \sum_{k'=1}^{n^{K'}} \operatorname{E}[\rho_{\mathbf{B}} \rho'_{\mathbf{B}}],$$

and

$$\mathrm{E}[\rho_n]\,\mathrm{E}[\rho_n'] = \frac{1}{n^{K+K'}} \sum_{k=1}^{n^K} \sum_{k'=1}^{n^{K'}} \mathrm{E}[\rho_{\mathbf{B}}]\,\mathrm{E}[\rho_{\mathbf{B}}'].$$

Hence, by letting $d_n = \mathrm{Cov}(\rho_n, \rho_n') = \mathrm{E}[\rho_n \rho_n'] - \mathrm{E}[\rho_n]\,\mathrm{E}[\rho_n']$,

$$\begin{aligned} d_n &= \frac{1}{n^{K+K'}} \sum_{k=1}^{n^K} \sum_{k'=1}^{n^{K'}} \big(\mathrm{E}[\rho_{\mathbf{B}}\rho_{\mathbf{B}}'] - \mathrm{E}[\rho_{\mathbf{B}}]\,\mathrm{E}[\rho_{\mathbf{B}}']\big) \\ &= \frac{1}{n^{K+K'}} \sum_{k=1}^{n^K} \sum_{k'=1}^{n^{K'}} \mathrm{Cov}(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}'). \end{aligned}$$

If $\mathbf{B}$ and $\mathbf{B}'$ share no blocks, then $\rho_{\mathbf{B}}$ and $\rho_{\mathbf{B}}'$ are independent and thus $\mathrm{Cov}(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}') = 0$. Thus we only need to consider the case that $\mathbf{B}$ and $\mathbf{B}'$ share at least one block. Similarly as before, we partition the pairs $(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}')$ based on the number of blocks $\mathbf{B}$ and $\mathbf{B}'$ share. According to Lemma A.3, for a fixed $S \in \mathcal{S}_r$, the number of pairs $(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}')$ such that $\rho_{\mathbf{B}} \sim_S \rho_{\mathbf{B}}'$ is

$$n^K (n-1)^{m-r} n^{K'-m} = n^{K+K'-m}(n-1)^{m-r}.$$

We hence have

$$\mathrm{Cov}(\rho_n, \rho_n') = \sum_{r=1}^{m} \frac{(n-1)^{m-r}}{n^m} \times \sum_{S \in \mathcal{S}_r} \mathrm{Cov}(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}').$$

Similarly as in Lemma A.4, we have

$$\mathrm{Cov}(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}') = \mathrm{E}_{\rho_{\mathbf{B}} \sim_S \rho_{\mathbf{B}}'}[(\rho_{\mathbf{B}} - \rho)(\rho_{\mathbf{B}}' - \rho')],$$

and hence

$$\mathrm{Cov}(\rho_{\mathbf{B}}, \rho_{\mathbf{B}}') = \frac{1}{|\Lambda(S)|} \sum_{l \in \Lambda(S)} \mathrm{E}_{\rho_{\mathbf{B}} \sim_{S(l)} \rho_{\mathbf{B}}'}[(\rho_{\mathbf{B}} - \rho)(\rho_{\mathbf{B}}' - \rho')].$$

Now let $\mathcal{K}$ and $\mathcal{K}'$ be the indexes of the relations in $\mathcal{R}$ and $\mathcal{R}'$ respectively. Denote $S_K^c = \mathcal{K} - S$ and $S_{K'}^c = \mathcal{K}' - S$. Let

$$E = E_{\rho_{\mathbf{B}} \sim S(\mathbf{l})\, \rho'_{\mathbf{B}}}[(\rho_{\mathbf{B}} - \rho)(\rho'_{\mathbf{B}} - \rho')].$$

We have

$$\begin{aligned} E &= \frac{1}{|\Lambda(S_K^c)| \cdot |\Lambda(S_{K'}^c)|} \sum_{\Lambda(S_K^c)} \sum_{\Lambda(S_{K'}^c)} \big((\rho_{\mathbf{B}} - \rho)(\rho'_{\mathbf{B}} - \rho')\big) \\ &= \big(\frac{1}{|\Lambda(S_K^c)|} \sum_{\Lambda(S_K^c)} (\rho_{\mathbf{B}} - \rho)\big)\big(\frac{1}{|\Lambda(S_{K'}^c)|} \sum_{\Lambda(S_{K'}^c)} (\rho'_{\mathbf{B}} - \rho')\big) \\ &= \big((\frac{1}{|\Lambda(S_K^c)|} \sum_{\Lambda(S_K^c)} \rho_{\mathbf{B}}) - \rho\big)\big((\frac{1}{|\Lambda(S_{K'}^c)|} \sum_{\Lambda(S_{K'}^c)} \rho'_{\mathbf{B}}) - \rho'\big) \\ &= (\rho_S(\mathbf{l}) - \rho)(\rho'_S(\mathbf{l}) - \rho'). \end{aligned}$$

This completes the proof of the lemma. □

Our second lemma further provides an upper bound for $\mathrm{Cov}_S(\rho, \rho')$:

Lemma A.7. *Let $S \in \mathcal{S}_r$. Then we have*

$$|\mathrm{Cov}_S(\rho, \rho')| \leqslant \sqrt{\sigma_S^2 \cdot (\sigma'_S)^2}.$$

Proof. Let $d_S = \mathrm{Cov}_S(\rho, \rho')$ and $d_\rho^2 = (\rho_S(\mathbf{l}) - \rho)^2$. We have

$$\begin{aligned} d_S^2 &= \frac{1}{|\Lambda(S)|^2}\big(\sum_{\mathbf{l} \in \Lambda(S)} (\rho_S(\mathbf{l}) - \rho)(\rho'_S(\mathbf{l}) - \rho')\big)^2 \\ &\leqslant \frac{1}{|\Lambda(S)|^2}\big(\sum_{\mathbf{l} \in \Lambda(S)} d_\rho^2\big)\big(\sum_{\mathbf{l} \in \Lambda(S)} d_{\rho'}^2\big) \\ &= \big(\frac{1}{|\Lambda(S)|} \sum_{\mathbf{l} \in \Lambda(S)} d_\rho^2\big)\big(\frac{1}{|\Lambda(S)|} \sum_{\mathbf{l} \in \Lambda(S)} d_{\rho'}^2\big) \\ &= \sigma_S^2 \cdot (\sigma'_S)^2, \end{aligned}$$

by the Cauchy-Schwarz inequality. It then follows that

$$|d_S| = |\mathrm{Cov}_S(\rho, \rho')| \leqslant \sqrt{\sigma_S^2 \cdot (\sigma'_S)^2},$$

which completes the proof of the lemma. □

We can now prove the first inequality in Theorem A.5:

Proof. Let $d_n = \mathrm{Cov}(\rho_n, \rho'_n)$. By Lemma A.6 and A.7 we have

$$\begin{aligned} |d_n| &= |\sum_{r=1}^{m} \big(1 - \frac{1}{n}\big)^{m-r} \big(\frac{1}{n}\big)^r \sum_{S \in \mathcal{S}_r} \mathrm{Cov}_S(\rho, \rho')| \\ &\leqslant \sum_{r=1}^{m} \big(1 - \frac{1}{n}\big)^{m-r} \big(\frac{1}{n}\big)^r \sum_{S \in \mathcal{S}_r} \sqrt{\sigma_S^2 (\sigma'_S)^2}. \end{aligned}$$

By the Cauchy-Schwarz inequality, we have

$$\sum_{S \in \mathcal{S}_r} \sqrt{\sigma_S^2 (\sigma'_S)^2} \leqslant \sqrt{\sum_{S \in \mathcal{S}_r} \sigma_S^2 \sum_{S \in \mathcal{S}_r} (\sigma'_S)^2}.$$

Combining these two inequalities, we obtain

$$|d_n| \leqslant \sum_{r=1}^{m} \big(1 - \frac{1}{n}\big)^{m-r} \big(\frac{1}{n}\big)^r \sqrt{\sum_{S \in \mathcal{S}_r} \sigma_S^2 \sum_{S \in \mathcal{S}_r} (\sigma'_S)^2}.$$

Now define

$$A_r = \sqrt{\big(1 - \frac{1}{n}\big)^{m-r} \big(\frac{1}{n}\big)^r \sum_{S \in \mathcal{S}_r} \sigma_S^2}.$$

Then it follows that

$$|d_n| \leqslant \sum_{r=1}^{m} A_r A'_r.$$

Applying the Cauchy-Schwarz inequality again,

$$|d_n| \leqslant \sqrt{\Big(\sum_{r=1}^{m} A_r^2\Big)\Big(\sum_{r=1}^{m} (A')_r^2\Big)} = \sqrt{S_\rho^2(m,n) S_{\rho'}^2(m,n)},$$

which completes the proof of the inequality. □

To establish the second inequality in the theorem, namely,

$$\sqrt{S_\rho^2(m,n) S_{\rho'}^2(m,n)} \leqslant \sqrt{\mathrm{Var}[\rho_n]\,\mathrm{Var}[\rho'_n]},$$

we need two more lemmas. The first one states that the σ_S^2 has some nice *monotonicity* property:

Lemma A.8. *Let $S \in \mathcal{S}_r$ and $S' \in \mathcal{S}_{r+1}$ such that $S \subset S'$, for $1 \leqslant r \leqslant K-1$. Then it holds that $\sigma_S^2 \leqslant \sigma_{S'}^2$.*

Proof. Without loss of generality, let $S = \{1,...,r\}$ and $S' = \{1,...,r+1\}$. For a given $\mathbf{l} = (j_1,...,j_r) \in \Lambda(S)$, let $\mathbf{l}'_j = (j_1,...,j_r,j)$, for $1 \leqslant j \leqslant m_{r+1}$. Since $\Lambda(S') = \Lambda(S) \times \{1,...,m_{r+1}\}$, we have $\Lambda(S^c) = \Lambda((S')^c) \times \{1,...,m_{r+1}\}$ and thus $|\Lambda(S^c)| = m_{r+1}|\Lambda((S')^c)|$. Therefore, by letting $d_\rho^2 = (\rho_S(\mathbf{l}) - \rho)^2$, it follows that

$$\begin{aligned}
d_\rho^2 &= \Big((\frac{1}{|\Lambda(S^c)|}\sum_{\Lambda(S^c)} \rho_{\mathbf{B}}) - \rho\Big)^2 \\
&= \Big(\frac{1}{|\Lambda(S^c)|}\sum_{\Lambda(S^c)} (\rho_{\mathbf{B}} - \rho)\Big)^2 \\
&= \Big(\frac{1}{m_{r+1}|\Lambda((S')^c)|}\sum_{j=1}^{m_{r+1}}\sum_{\Lambda((S')^c)} (\rho_{\mathbf{B}} - \rho)\Big)^2 \\
&= \frac{1}{m_{r+1}^2}\Big(\sum_{j=1}^{m_{r+1}} \frac{1}{|\Lambda((S')^c)|}\sum_{\Lambda((S')^c)} (\rho_{\mathbf{B}} - \rho)\Big)^2.
\end{aligned}$$

By the Cauchy-Schwarz inequality, we have

$$\begin{aligned} d_\rho^2 &= \frac{1}{m_{r+1}} \sum_{j=1}^{m_{r+1}} \Big(\frac{1}{|\Lambda((S')^c)|} \sum_{\Lambda((S')^c)} (\rho_{\mathbf{B}} - \rho) \Big)^2 \\ &= \frac{1}{m_{r+1}} \sum_{j=1}^{m_{r+1}} \Big((\frac{1}{|\Lambda((S')^c)|} \sum_{\Lambda((S')^c)} \rho_{\mathbf{B}}) - \rho \Big)^2 \\ &= \frac{1}{m_{r+1}} \sum_{j=1}^{m_{r+1}} (\rho_{S'}(\mathbf{1}'_j) - \rho)^2. \end{aligned}$$

Therefore,

$$\begin{aligned} \sigma_S^2 &= \frac{1}{|\Lambda(S)|} \sum_{\Lambda(S)} (\rho_S(\mathbf{1}) - \rho)^2 \\ &\leqslant \frac{1}{|\Lambda(S)| m_{r+1}} \sum_{\Lambda(S)} \sum_{j=1}^{m_{r+1}} (\rho_{S'}(\mathbf{1}'_j) - \rho)^2 \\ &= \frac{1}{|\Lambda(S')|} \sum_{\Lambda(S')} (\rho_{S'}(\mathbf{1}') - \rho)^2 \\ &= \sigma_{S'}^2. \end{aligned}$$

This completes the proof of the lemma. □

Our next lemma shows that the $S_\rho^2(m, n)$ also has some monotonicity property:

Lemma A.9. *For $m \geqslant 1$, we have*

$$S_\rho^2(m, n) \leqslant S_\rho^2(m + 1, n).$$

Proof. We should be careful now since $\mathcal{S}_r$ is actually related to m. Specifically, $\mathcal{S}_r$ is all the r-subsets of $\{1, ..., m\}$.[1] To make this more explicit, we further use $\mathcal{S}_r^{(m)}$ to

[1] More generally, the indexes could be represented as $\mathcal{J}_m = \{j_1, ..., j_m\}$ and $\mathcal{J}_{m+1} = \{j_1, ..., j_m, j_{m+1}\}$ such that $\mathcal{J}_m \subset \mathcal{J}_{m+1}$. We used $\mathcal{J}_m = \{1, ..., m\}$ and $\mathcal{J}_{m+1} = \{1, ..., m, m + 1\}$ in our proof without loss of generality.

indicate this relationship. Moreover, to simplify notation, we define

$$A_r^{(m)} = \sum_{S \in \mathcal{S}_r^{(m)}} \sigma_S^2.$$

Furthermore, if $r = m$, then $\mathcal{S}_m^{(m)}$ contains only one single element $\{1, ..., m\}$. We thus simply use σ_m^2 to represent $A_m^{(m)}$, i.e.,

$$\sigma_m^2 = \sum_{S \in \mathcal{S}_m^{(m)}} \sigma_S^2.$$

Now consider $S_{m+1} = S_\rho^2(m+1, n)$. We have

$$\begin{aligned} S_{m+1} &= \sum_{r=1}^{m+1} \left(1 - \frac{1}{n}\right)^{m+1-r} \left(\frac{1}{n}\right)^r A_r^{(m+1)} \\ &= \sum_{r=1}^{m} \left(1 - \frac{1}{n}\right)^{m+1-r} \left(\frac{1}{n}\right)^r A_r^{(m+1)} + \left(\frac{1}{n}\right)^{m+1} \sigma_{m+1}^2. \end{aligned}$$

Define $\Delta_r^{(m+1)} = \sum_{S \in \mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}} \sigma_S^2$. Then

$$\Delta_r^{(m+1)} = A_r^{(m+1)} - A_r^{(m)}.$$

We therefore have $S_{m+1} = \left(1 - \frac{1}{n}\right) S_m + B_m$, where

$$B_m = \sum_{r=1}^{m} \left(1 - \frac{1}{n}\right)^{m+1-r} \left(\frac{1}{n}\right)^r \Delta_r^{(m+1)} + \left(\frac{1}{n}\right)^{m+1} \sigma_{m+1}^2.$$

Let us further define $\mathcal{S}_r^{(m)} = \emptyset$ if $r > m$. Then $\Delta_{m+1}^{(m+1)} = \sigma_{m+1}^2$, and therefore

$$B_m = \sum_{r=1}^{m+1} \left(1 - \frac{1}{n}\right)^{m+1-r} \left(\frac{1}{n}\right)^r \Delta_r^{(m+1)}.$$

Next, consider some $S \in \mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}$ where $r \geqslant 2$. Note that S must contain $m+1$ since otherwise $S \in \mathcal{S}_r^{(m)}$. What's more, if we remove $m+1$ from S, then S must be now in $\mathcal{S}_{r-1}^{(m)}$, that is, $S \setminus \{m+1\} \in \mathcal{S}_{r-1}^{(m)}$. On the other hand, for any $S' \in \mathcal{S}_{r-1}^{(m)}$, we can obtain an element in $\mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}$ by simply adding $m+1$, that is, $S' \cup \{m+1\} \in \mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}$. We therefore have established a 1-1 mapping φ between $\mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}$ and $\mathcal{S}_{r-1}^{(m)}$.

Furthermore, note that for any $S \in \mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}$, we have $\varphi(S) \subset S$. Hence by Lemma A.8, $\sigma^2_{\varphi(S)} \leqslant \sigma^2_S$. Therefore, we have

$$\Delta_r^{(m+1)} = \sum_{S \in \mathcal{S}_r^{(m+1)} \setminus \mathcal{S}_r^{(m)}} \sigma_S^2 \geqslant \sum_{\varphi(S) \in \mathcal{S}_{r-1}^{(m)}} \sigma^2_{\varphi(S)} = A_{r-1}^{(m)}.$$

As a result, we have

$$\begin{aligned} B_m &\geqslant C_m + \sum_{r=2}^{m+1} \big(1 - \frac{1}{n}\big)^{m+1-r} \big(\frac{1}{n}\big)^r A_{r-1}^{(m)} \\ &= C_m + \sum_{r'=1}^{m} \big(1 - \frac{1}{n}\big)^{m+1-(r'+1)} \big(\frac{1}{n}\big)^{r'+1} A_{r'}^{(m)} \\ &= C_m + \frac{1}{n} \sum_{r'=1}^{m} \big(1 - \frac{1}{n}\big)^{m-r'} \big(\frac{1}{n}\big)^{r'} A_{r'}^{(m)} \\ &= C_m + \frac{1}{n} S_m, \end{aligned}$$

where

$$C_m = \big(1 - \frac{1}{n}\big)^m \frac{1}{n} \Delta_1^{(m+1)} = \frac{1}{n}\big(1 - \frac{1}{n}\big)^m \sigma^2_{\{m+1\}} \geqslant 0.$$

Hence, $B_m \geqslant \frac{1}{n} S_m$. Since $S_{m+1} = \big(1 - \frac{1}{n}\big) S_m + B_m$, we conclude that $S_{m+1} \geqslant S_m$. This completes the proof of the lemma. □

It is now easy to prove the second inequality in Theorem A.5:

Proof. Based on Lemma A.9, by induction, we can easily prove that $S^2_\rho(m, n) \leqslant \mathrm{Var}[\rho_n]$ and $S^2_{\rho'}(m, n) \leqslant \mathrm{Var}[\rho'_n]$, since $m \leqslant \min\{K, K'\}$. The second inequality in Theorem A.5 then follows. □

For our special case in this dissertation where $\mathrm{Cov}(\rho_n, \rho'_n) \neq 0$, we will always have $m = \min\{K, K'\}$. Without loss of generality, let $m = K$. Then $S^2_\rho(m, n) = \mathrm{Var}[\rho_n]$, and we only need to approximate $S^2_{\rho'}(m, n)$ with $S^2_{\rho'}(K, n)$, which by Lemma A.9 is guaranteed to be superior to $\mathrm{Var}[\rho'_n]$. Intuitively, the bigger $K' - K$ is, the bigger the gap is between $S^2_{\rho'}(K, n)$ and $\mathrm{Var}[\rho'_n]$. In fact, in the proof of Lemma A.9, we have actually showed that $S_{m+1} \geqslant S_m + C_m$. So we can roughly estimate that

$$\mathrm{Var}[\rho'_n] - S^2_{\rho'}(K, n) \geqslant \frac{1}{n}\big(1 - \frac{1}{n}\big)^K \sum_{r=K+1}^{K'} \sigma^2_{\{r\}}.$$

A.4 More Discussions on Covariances

We can actually have another upper bound for $\mathrm{Cov}(\rho_n, \rho'_n)$:

Theorem A.10. *We have*

$$|\mathrm{Cov}(\rho_n, \rho'_n)| \leqslant f(n, m) g(\rho) g(\rho'),$$

where $f(n, m) = 1 - (1 - \frac{1}{n})^m$ *and* $g(\rho) = \sqrt{\rho(1-\rho)}$.

Proof. As in the proof of Lemma A.6, let $\mathcal{K}$ and $\mathcal{K}'$ be the indexes of the relations in $\mathcal{R}$ and $\mathcal{R}'$. By Lemma A.8, we have $\sigma^2_S \leqslant \sigma^2_{\mathcal{K}}$ and $(\sigma'_S)^2 \leqslant \sigma^2_{\mathcal{K}'}$. Moreover, consider

$$\sigma^2_{\mathcal{K}} = \frac{1}{|\Lambda(\mathcal{K})|} \sum_{\mathbf{l} \in \Lambda(\mathcal{K})} (\rho_{\mathcal{K}}(\mathbf{l}) - \rho)^2.$$

Since we use the tuple-level partition scheme, we have $\rho_{\mathcal{K}}(\mathbf{l}) = 1$ or $\rho_{\mathcal{K}}(\mathbf{l}) = 0$. Therefore, it follows that

$$\begin{aligned}
\sigma^2_{\mathcal{K}} &= (1-\rho)^2 \cdot \frac{1}{|\Lambda(\mathcal{K})|} \sum_{\mathbf{l} \in \Lambda(\mathcal{K})} I(\rho_{\mathcal{K}}(\mathbf{l}) = 1) + \rho^2 \cdot \frac{1}{|\Lambda(\mathcal{K})|} \sum_{\mathbf{l} \in \Lambda(\mathcal{K})} I(\rho_{\mathcal{K}}(\mathbf{l}) = 0) \\
&= (1-\rho)^2 \cdot \rho + \rho^2 \cdot (1-\rho) \\
&= \rho(1-\rho).
\end{aligned}$$

Similarly, we have $\sigma^2_{\mathcal{K}'} = \rho'(1-\rho')$. Hence,

$$\mathrm{Cov}_S(\rho,\rho') \leqslant \sqrt{\sigma^2_S(\sigma'_S)^2} \leqslant \sqrt{\rho(1-\rho)\cdot\rho'(1-\rho')},$$

and therefore, by letting $g(\rho) = \sqrt{\rho(1-\rho)}$,

$$\begin{aligned}
|\mathrm{Cov}(\rho_n,\rho'_n)| &= |\sum_{r=1}^{m} \frac{(n-1)^{m-r}}{n^m} \times \sum_{S\in\mathcal{S}_r} \mathrm{Cov}_S(\rho,\rho')| \\
&\leqslant \sum_{r=1}^{m} \frac{(n-1)^{m-r}}{n^m} \times \sum_{S\in\mathcal{S}_r} g(\rho)g(\rho') \\
&= g(\rho)g(\rho')\sum_{r=1}^{m}\binom{m}{r}(\frac{1}{n})^r(1-\frac{1}{n})^{m-r} \\
&= g(\rho)g(\rho')[1-(1-\frac{1}{n})^m].
\end{aligned}$$

This completes the proof of the theorem. □

When n is large, $(1-\frac{1}{n})^m \approx 1-\frac{m}{n}$. As a result, $1-(1-\frac{1}{n})^m \approx \frac{m}{n}$. Therefore, when $n \to \infty$, $\mathrm{Cov}(\rho_n,\rho'_n) \to 0$. This is intuitively true considering the strong consistency of ρ_n. If we keep taking samples, finally the estimated selectivity should converge to the actual selectivity (a constant). On the other hand, a larger m implies a larger bound since the computations of ρ_n and ρ'_n share more samples. Another interesting observation is that the bound also depends on the actual selectivities ρ and ρ'. Note that $g(\rho)$ is minimized at $\rho = 0$ or $\rho = 1$ (with $g_{\min} = 0$), and is maximized at $\rho = \frac{1}{2}$ (with $g_{\max} = \frac{1}{2}$). To shed some light on this, observe that whenever ρ or ρ' is 0 or 1, ρ_n or ρ'_n is always 0 or 1 regardless of the number of samples. Hence $\mathrm{Cov}(\rho_n,\rho'_n) = 0$ in such cases.

A natural question is how good this bound is compared with the two bounds in Section 4.5.3.2. Let us name these three bounds as:

(B_1) $\sqrt{S^2_\rho(m,n)S^2_{\rho'}(m,n)}$, the first bound in Theorem A.5;

(B_2) $\sqrt{\mathrm{Var}[\rho_n]\,\mathrm{Var}[\rho'_n]}$, the second bound in Theorem A.5;

(B_3) $f(n, m)g(\rho)g(\rho')$, the bound in Theorem A.10.

By Theorem A.5, we already know that $B_1 \leqslant B_2$. Next, according to the proof of Theorem A.10, $\sigma_S^2 \leqslant \rho(1-\rho)$ and $(\sigma')_S^2 \leqslant \rho'(1-\rho')$. We then immediately have

$$\sqrt{S_\rho^2(m,n)S_{\rho'}^2(m,n)} \leqslant f(n,m)g(\rho)g(\rho'),$$

by the definition of $S_\rho^2(m,n)$. That is, $B_1 \leqslant B_3$. Moreover, by Theorem 4.9, we have

$$|\mathrm{Cov}(\rho_n, \rho_n')| \leqslant \sqrt{\mathrm{Var}[\rho_n]\,\mathrm{Var}[\rho_n']} \leqslant f(n)g(\rho)g(\rho'),$$

where

$$f(n) = \sqrt{\big(1-(1-\frac{1}{n})^K\big)\big(1-(1-\frac{1}{n})^{K'}\big)}.$$

When n is large, $1-(1-\frac{1}{n})^K \approx \frac{K}{n}$, and $1-(1-\frac{1}{n})^{K'} \approx \frac{K'}{n}$. Therefore, the right hand is close to $\frac{\sqrt{KK'}}{n}g(\rho)g(\rho')$. Since $m \leqslant \min\{K, K'\} < \sqrt{KK'}$, we know that B_3 is better than the upper bound of B_2. However, in general B2 and B_3 are incomparable.

One more issue of B_3 is that it includes the *true* selectivities ρ and ρ' that are not known without running the query. As a result, B_3 is not directly computable. Nonetheless, when n is large, we can simply use the observed ρ_n and ρ_n' as approximations due to the strong consistency of ρ_n.

A.5 Proof of Lemma 5.8

Proof. The procedure terminates when the head ball is marked. Since a marked ball is uniformly placed at any position in the queue, the probability that a ball is marked after k steps is k/N ($1 \leqslant k \leqslant N$). So is the probability that the head ball is marked. Formally, let A_k be the event that the head ball is marked after exactly k steps. Then $\Pr(A_1) = 1/N$ and $\Pr(A_k|\bar{A}_1 \cap \cdots \cap \bar{A}_{k-1}) = k/N$. As a result, letting

$B_k = \bar{A}_1 \cap \cdots \cap \bar{A}_{k-1}$ we have

$$\begin{aligned} \Pr(A_k) &= \Pr(A_k|B_k)\Pr(B_k) + \Pr(A_k|\bar{B}_k)\Pr(\bar{B}_k) \\ &= \frac{k}{N} \cdot \Pr(B_k) + 0 \cdot \Pr(\bar{B}_k) \\ &= \frac{k}{N} \cdot \Pr(B_k). \end{aligned}$$

Now let us calculate $\Pr(B_k)$. We have

$$\begin{aligned} \Pr(B_k) &= \Pr(\bar{A}_1 \cap \cdots \cap \bar{A}_{k-1}) \\ &= \Pr(\bar{A}_{k-1}|\bar{A}_1 \cap \cdots \cap \bar{A}_{k-2})\Pr(\bar{A}_1 \cap \cdots \cap \bar{A}_{k-2}) \\ &= (1 - \frac{k-1}{N})\Pr(B_{k-1}). \end{aligned}$$

Therefore, we now have a recurrence equation for $\Pr(B_k)$ and thus,

$$\begin{aligned} \Pr(B_k) &= (1 - \frac{k-1}{N}) \cdots (1 - \frac{2}{N})\Pr(B_2) \\ &= (1 - \frac{k-1}{N}) \cdots (1 - \frac{2}{N})\Pr(\bar{A}_1) \\ &= (1 - \frac{k-1}{N}) \cdots (1 - \frac{2}{N})(1 - \frac{1}{N}). \end{aligned}$$

As a result, the expected number of steps that the procedure would take before its termination is then

$$\begin{aligned} S_N &= \sum_{k=1}^{N} k \cdot \Pr(A_k) \\ &= \sum_{k=1}^{N} k \cdot (1 - \frac{1}{N}) \cdots (1 - \frac{k-1}{N}) \cdot \frac{k}{N}. \end{aligned}$$

This completes the proof of the lemma. □

A.6 Proof of Theorem 5.9

We need the following lemma before we prove Theorem 5.9.

Lemma A.11. *Let the* k*-th summand in* S_N *be* X_k*, i.e.,*

$$\begin{aligned} X_k &= k \cdot (1 - \frac{1}{N}) \cdots (1 - \frac{k-1}{N}) \cdot \frac{k}{N} \\ &= \frac{N!}{(N-k)!} \cdot \frac{1}{N^k} \cdot \frac{k^2}{N}. \end{aligned}$$

For any $k \geqslant N^{1/2+\epsilon}$ *(*$\epsilon > 0$*), we have* $X_k = O(e^{-N^{2\epsilon}})$.

Proof. We prove this in two steps:

(i) if $N^{1/2+\epsilon} \leqslant k \leqslant N/2$, then $X_k = O(e^{-N^{2\epsilon}})$;

(ii) if $k \geqslant N/2$, then $X_{k+1} < X_k$ for sufficiently large N ($N \geqslant 5$ indeed).

We prove (i) first. Consider $\ln X_k$. We have

$$\ln X_k = \ln\big(N!\big) - \ln\big((N-k)!\big) + 2\ln k - (k+1)\ln N.$$

By using Stirling's formula,

$$\ln\big(N!\big) = N\ln N - N + O(\ln N).$$

It then follows that

$$\begin{aligned} \ln X_k &= (N-k)\ln N - (N-k)\ln(N-k) - k + O(\ln N) \\ &= (N-k)\ln\big(\frac{N}{N-k}\big) - k + O(\ln N) \\ &= (N-k)\ln\big(1 + \frac{k}{N-k}\big) - k + O(\ln N). \end{aligned}$$

Using Taylor's formula for $f(x) = \ln(1+x)$, we have

$$\ln\big(1+\frac{k}{N-k}\big) = \frac{k}{N-k} - \frac{1}{2}\cdot\frac{k^2}{(N-k)^2} + O\big(\frac{1}{3}\cdot\frac{k^3}{(N-k)^3}\big).$$

Therefore, it follows that

$$\ln X_k = -\frac{1}{2}\cdot\frac{k^2}{N-k} + O\big(\frac{1}{3}\cdot\frac{k^3}{(N-k)^2}\big) + O(\ln N).$$

Since $k \leqslant N/2$, $\frac{k}{N-k} \leqslant 1$. As a result,

$$\frac{k^3}{(N-k)^2} = \frac{k}{N-k}\cdot\frac{k^2}{N-k} \leqslant \frac{k^2}{N-k}.$$

On the other hand, since $k \geqslant N^{1/2+\epsilon}$, $k^2 \geqslant N^{1+2\epsilon}$ and thus,

$$\frac{k^2}{N-k} > \frac{k^2}{N} \geqslant N^{2\epsilon} > O(\ln N).$$

Therefore, $\ln X_k$ is dominated by the term $\frac{k^2}{N-k}$. Hence

$$\ln X_k = O(-\frac{k^2}{N-k}) = O(-N^{2\epsilon}),$$

which implies $X_k = O(e^{-N^{2\epsilon}})$.

We next prove (ii). We consider the ratio of X_{k+1} and X_k, which gives

$$r_k = \frac{X_{k+1}}{X_k} = \frac{N-k}{N}\cdot\frac{(k+1)^2}{k^2} = \big(1-\frac{k}{N}\big)\cdot\big(1+\frac{1}{k}\big)^2.$$

Since $k \geqslant N/2$, $1-\frac{k}{N} \leqslant 1/2$ and $\frac{1}{k} \leqslant 2/N$. It follows that

$$r_k \leqslant \frac{1}{2}\cdot\big(1+\frac{2}{N}\big)^2.$$

Letting $r_k < 1$ gives $N > 2(\sqrt{2}+1) \approx 4.83$. So we have $r_k < 1$ when $N \geqslant 5$, which concludes the proof of the lemma. □

We are ready to give a proof to Theorem 5.9.

Proof. (of Theorem 5.9) Let X_k be the same as defined in Lemma A.11. Then

$$S_N = \sum_{k=1}^{N} X_k.$$

According to Lemma A.11, $X_k = O(e^{-N^{2\epsilon}})$ if $k \geqslant N^{1/2+\epsilon}$. It then follows that

$$\sum\nolimits_{k \geqslant N^{1/2+\epsilon}} X_k = O(N \cdot e^{-N^{2\epsilon}}) = o(\sqrt{N}).$$

On the other hand, when $k < N^{1/2+\epsilon}$, we can pick a sufficiently small ϵ so that $k \leqslant N^{1/2}$. If this is the case, then $k^2 \leqslant N$ and thus $\frac{k^2}{N} \leqslant 1$. As a result, $X_k \leqslant 1$ and hence we have

$$\sum\nolimits_{k < N^{1/2+\epsilon}} X_k \leqslant \sqrt{N}.$$

It follows that

$$S_N = \sum\nolimits_{k \geqslant N^{1/2+\epsilon}} X_k + \sum\nolimits_{k < N^{1/2+\epsilon}} X_k = O(\sqrt{N}),$$

which completes the proof of the theorem.

To pick such an ϵ, note that it is sufficient if ϵ satisfies $N^{1/2+\epsilon} \leqslant \lfloor N^{1/2} \rfloor + 1$. It then follows that

$$\epsilon \leqslant \ln\left(1 + \lfloor \sqrt{N} \rfloor\right) / \ln N - 1/2.$$

Note that the right hand side must be greater than 0, because $\ln\left(1+\lfloor\sqrt{N}\rfloor\right) > \ln\sqrt{N}$. As an example, if $N = 100$, then $\epsilon \leqslant 0.0207$. □

A.7 Additional Analysis of Re-Optimization

Continuing with our study of the efficiency of the re-optimization procedure in Section 5.3.3, we now consider two special cases where the optimizer significantly overestimates or underestimates the selectivity of a particular join in the returned optimal plan. We focus our discussion on left-deep join trees. Before we proceed, we need the following assumption of cost functions.

Assumption 3. *Cost functions are monotonically non-decreasing functions with respect to input cardinalities.*

We are unaware of cost functions that do not conform to this assumption, though.

A.7.1 Overestimation Only

Suppose that the error is an overestimate, that is, the actual cardinality is smaller than the one estimated by the optimizer. Let that join operator be O, and let the corresponding subtree rooted at O be $\text{tree}(O)$. Now consider a candidate plan P in the search space, and let $\text{cost}(P)$ be the estimated cost of P. Note that the following property holds:

Lemma A.12. $\text{cost}(P)$ *is subject to change only if* $\text{tree}(O)$ *is a subtree of* $\text{tree}(P)$.

Moreover, under Assumption 3, the refined cost estimate $\text{cost}'(P)$ satisfies $\text{cost}'(P) < \text{cost}(P)$ because of the overestimate. Therefore, if we let the set of all such plans P be $\mathcal{P}$, then the next optimal plan picked by the optimizer must be from $\mathcal{P}$. We thus can prove the following result:

Theorem A.13. *Suppose that we only consider left-deep join trees. Let* m *be the number of joins in the query. If all estimation errors are overestimates, then in the worst case the re-optimization procedure would terminate in at most* $m + 1$ *steps.*

Proof. We use P_i to denote the plan returned in the i-th step, and use O_i to denote the *lowest* join in P_i where an overestimate occurs. We use $I(O_i)$ to denote the index of O_i in $\text{tree}(P_i)$ by ordering the joins in a bottom-up, left-to-right manner.

Let $\mathcal{P}_i$ be the plans whose join trees contain tree(O_i) as a subtree. By Lemma A.12, $P_{i+1} \in \mathcal{P}_i$. Moreover, we must have $I(O_{i+1}) > I(O_i)$, that is, the lowest join in P_{i+1} with an overestimate must be at a higher level than that in P_i, because tree(O_i) is a subtree of tree(P_{i+1}) and tree(O_i) has been validated when validating P_i. Note that we can only have at most m different $I(O_i)$'s given that the query contains m joins. Therefore, after (at most) m steps, we must have $\mathcal{P}_m = \emptyset$. As a result, by Theorem 5.7 we can only have at most one additional *local* transformation on top of P_m. But this cannot occur for left-deep trees, because tree(P_{m-1}) = tree(P_m) if the worst case really occurs (i.e., in every step an overestimate occurs such that $I(O_{i+1}) = I(O_i) + 1$). So any local transformation must have been considered when generating P_m by using validated cardinalities from P_{m-1}. Hence the re-optimization procedure terminates in at most $m + 1$ steps: the $(m + 1)$-th step just checks $P_{m+1} = P_m$, which must be true. □

We emphasize that $m+1$ is a worst-case upper bound only. The intuition behind Theorem A.13 is that, for left-deep join trees, the validated subtree belonging to the final re-optimized plan can grow by at least one more level (i.e., with one more validated join) in each re-optimization step.

A.7.2 Underestimation Only

Suppose that, on the other hand, the error is an underestimate, that is, the actual cardinality is larger than the one estimated by the optimizer. Then things become more complicated: not only those plans that contain the subtree rooted at the erroneous node but the plan with the lowest estimated cost in the rest of the search space (which is not affected by the estimation error) also has the potential of being the optimal plan, after the error is corrected (see Figure A.1). We next again focus on left-deep trees and present some analysis in such context.

Suppose that the join graph G contains M edges. We partition the join trees based on their first joins, which must be the edges of G. So we have M partitions.

Let s_i be the state when a plan returned by the optimizer uses the i-th edge in G ($1 \leqslant i \leqslant M$) as its first join. The re-optimization procedure can then be thought of

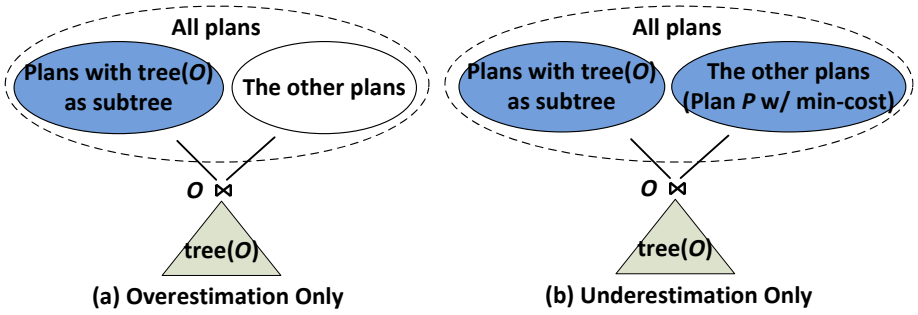


Figure A.1: Comparison of the space of candidate optimal plans when local estimation errors are overestimation-only or underestimation-only. Candidate optimal plans are shaded.

as transitions between these M states, i.e., a Markov chain. As before, we assume that the transition probabilities are uniform, i.e., for $1 \leqslant i, j \leqslant M$,

$$\pi(s_j|s_i) = 1/M.$$

In other words, it is equally likely that the next plan generated would have its first join in the partition i ($1 \leqslant i \leqslant M$). Then the equilibrium state distribution of this Markov chain is also uniform. That is, given any plan generated in the re-optimization procedure, it is equally likely that it would have its first join in the partition i ($1 \leqslant i \leqslant M$). We can estimate the expected number of steps before the procedure terminates as

$$S = \sum_{i=1}^{M} \pi(s_i) \cdot N_i,$$

where N_i is the number of steps/transitions confined in the partition i before termination. Since we only consider left-deep trees, $N_i = N_j$ for $1 \leqslant i, j \leqslant M$. As a result, S could be simplified as $S = N_i$.

As an upper bound, we have $S \leqslant S_{N/M}$, where N is the total number of different join trees considered by the optimizer, and $S_{N/M}$ is computed by Equation (5.1). Note that $S_{N/M}$ is usually much smaller than S_N, according to Figure 5.3. Again, we emphasize that the analysis here only targets worst-case expectations, which might be too pessimistic.

References

[1] http://infolab.stanford.edu/~widom/cs346/db2-talk.pdf.

[2] docs.oracle.com/cd/B10500_01/appdev.920/a96595/dci08opt.htm.

[3] http://www.qdpma.com/CBO/SQLServerCostBasedOptimizer.html.

[4] http://www.postgresql.org/docs/9.0/static/view-pg-stats.html.

[5] Skewed TPC-H data generator. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.

[6] TPC-H benchmark. http://www.tpc.org/tpch/.

[7] Abiteboul, Serge, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Addison-Wesley.

[8] Acharya, Swarup, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *SIGMOD*, 275–286.

[9] Ahmad, Mumtaz, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2011. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal* 20:589–615.

[10] Ahmad, Mumtaz, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. 2011. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *EDBT*, 449–460.

[11] Akdere, Mert, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based query performance modeling and prediction. In *ICDE*, 390–401.

[12] Alon, Noga, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking join and self-join sizes in limited storage. In *PODS*, 10–20.

[13] Aroian, Leo A. 1947. The probability function of the product of two normally distributed variables. *Ann. Math. Statist* 18(2):265–271.

[14] Babcock, Brian, and Surajit Chaudhuri. 2005. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*, 119–130.

[15] Bach, Francis R., and Michael I. Jordan. 2002. Kernel independent component analysis. *Journal of Machine Learning Research* 3:1–48.

[16] Breitling, Wolfgang. Joins, skew and histograms. `http://www.centrexcc.com/Joins,SkewandHistograms.pdf`.

[17] Bruno, Nicolas, and Surajit Chaudhuri. 2002. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 263–274.

[18] Bruno, Nicolas, Surajit Chaudhuri, and Luis Gravano. 2001. Stholes: A multidimensional workload-aware histogram. In *SIGMOD*, 211–222.

[19] Charikar, Moses, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 2000. Towards estimation error guarantees for distinct values. In *PODS*, 268–279.

[20] Chaudhuri, Surajit. 1998. An overview of query optimization in relational systems. In *PODS*, 34–43.

[21] Chaudhuri, Surajit, Raghav Kaushik, and Ravishankar Ramamurthy. 2005. When can we trust progress estimators for sql queries? In *SIGMOD*, 575–586.

[22] Chaudhuri, Surajit, Rajeev Motwani, and Vivek R. Narasayya. 1999. On random sampling over joins. In *SIGMOD*, 263–274.

[23] Chaudhuri, Surajit, Vivek R. Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for SQL queries. In *SIGMOD*, 803–814.

[24] ———. 2008. Diagnosing estimation errors in page counts using execution feedback. In *ICDE*, 1013–1022.

[25] ———. 2008. A pay-as-you-go framework for query execution feedback. *PVLDB* 1(1):1141–1152.

[26] Chi, Yun, Hakan Hacigümüs, Wang-Pin Hsiung, and Jeffrey F. Naughton. 2013. Distribution-based query scheduling. *PVLDB* 6(9):673–684.

[27] Chi, Yun, Hyun Jin Moon, and Hakan Hacıgümüş. 2011. iCBS: Incremental costbased scheduling under piecewise linear slas. *PVLDB* 4(9):563–574.

[28] Chu, Francis C., Joseph Y. Halpern, and Praveen Seshadri. 1999. Least expected cost query optimization: An exercise in utility. In *PODS*, 138–147.

[29] Cole, Richard L., and Goetz Graefe. 1994. Optimization of dynamic query evaluation plans. In *SIGMOD*, 150–160.

[30] D., Harish, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *PVLDB* 1(1):1124–1140.

[31] Devlin, Susan J., R. Gnanadesikan, and J. R. Kettenring. 1975. Robust estimation and outlier detection with correlation coefficients. *Biometrika* 62(3):531–545.

[32] Du, Weimin, Ravi Krishnamurthy, and Ming-Chien Shan. 1992. Query optimization in a heterogeneous dbms. In *VLDB*, 277–291.

[33] Duggan, Jennie, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *SIGMOD*, 337–348.

[34] Dutt, Anshuman, and Jayant R. Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, 1039–1050.

[35] Friedman, Jerome H. 2002. Stochastic gradient boosting. *Comput. Stat. Data Anal.* 38(4):367–378.

[36] Ganapathi, Archana, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 592–603.

[37] Getoor, Lise, Benjamin Taskar, and Daphne Koller. 2001. Selectivity estimation using probabilistic models. In *SIGMOD*, 461–472.

[38] Graefe, Goetz. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25(2):73–170.

[39] ———. 2011. Robust query processing. In *ICDE*, 1361.

[40] Graefe, Goetz, and Karen Ward. 1989. Dynamic query evaluation plans. In *SIGMOD*, 358–366.

[41] Green, Todd J., Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*, 31–40.

[42] Guirguis, Shenoda, Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. 2009. Adaptive scheduling of web transactions. In *ICDE*, 357–368.

[43] Haas, Peter J., Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. 1995. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, 311–322.

[44] Haas, Peter J., Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci.* 52(3):550–569.

[45] Haas, Peter J., and Arun N. Swami. 1992. Sequential sampling procedures for query size estimation. In *SIGMOD*, 341–350.

[46] Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explorations* 11(1):10–18.

[47] Hastie, T., R. Tibshirani, and J. H. Friedman. 2003. *The Elements of Statistical Learning*. Springer.

[48] Hellerstein, Joseph M., Peter J. Haas, and Helen J. Wang. 1997. Online aggregation. In *SIGMOD*, 171–182.

[49] Hou, Wen-Chi, and Gultekin Ozsoyoglu. 1991. Statistical estimators for aggregate relational algebra queries. *ACM Trans. Database Syst.* 16.

[50] Hou, Wen-Chi, Gultekin Özsoyoglu, and Baldeo K. Taneja. 1988. Statistical estimators for relational algebra expressions. In *PODS*, 276–287.

[51] Ilyas, Ihab F., Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 647–658.

[52] Ioannidis, Yannis E. 1996. Query optimization. *ACM Comput. Surv.* 28(1): 121–123.

[53] ———. 2003. The history of histograms (abridged). In *VLDB*, 19–30.

[54] Ioannidis, Yannis E., Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1992. Parametric query optimization. In *VLDB*, 103–114.

[55] Jermaine, Christopher M., Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2007. Scalable approximate query processing with the dbo engine. In *SIGMOD*, 725–736.

[56] Kabra, Navin, and David J. DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 106–117.

[57] König, Arnd Christian, Bolin Ding, Surajit Chaudhuri, and Vivek R. Narasayya. 2011. A statistical approach towards robust progress estimation. *PVLDB* 5(4): 382–393.

[58] Larson, Per-Åke, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *SIGMOD*, 175–186.

[59] Lazowska, Edward D., John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall.

[60] Li, Jiexing, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for sql queries using statistical techniques. *PVLDB* 5(11):1555–1566.

[61] Li, Jiexing, Rimma V. Nehme, and Jeffrey F. Naughton. 2012. GSLPI: A cost-based query progress indicator. In *ICDE*, 678–689.

[62] Lipton, Richard J., Jeffrey F. Naughton, and Donovan A. Schneider. 1990. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, 1–11.

[63] Lohman, Guy. Is query optimization a "solved" problem? `http://wp.sigmod.org/?p=1075`.

[64] Luo, Gang, Jeffrey F. Naughton, Curt J. Ellmann, and Michael Watzke. 2004. Toward a progress indicator for database queries. In *SIGMOD*, 791–802.

[65] Manegold, Stefan, Peter A. Boncz, and Martin L. Kersten. 2002. Generic database cost models for hierarchical memory systems. In *VLDB*, 191–202.

[66] Markl, Volker, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. 2004. Robust query processing through progressive optimization. In *SIGMOD*, 659–670.

[67] Mishra, Chaitanya, and Nick Koudas. 2009. The design of a query monitoring system. *ACM Trans. Database Syst.* 34(1).

[68] Muralikrishna, M., and David J. DeWitt. 1988. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *SIGMOD*, 28–36.

[69] Myers, Jerome L., and Arnold D. Well. 2003. *Research design and statistical analysis*. 2nd ed. Lawrence Erlbaum.

[70] Nicola, Victor F., Asit Dan, and Daniel M. Dias. 1992. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *SIGMETRICS*, 35–46.

[71] Osman, Rasha, Irfan Awan, and Michael E. Woodward. 2009. Application of queueing network models in the performance evaluation of database designs. *Electr. Notes Theor. Comput. Sci.* 232:101–124.

[72] Patel, Jignesh M., Michael J. Carey, and Mary K. Vernon. 1994. Accurate modeling of the hybrid hash join algorithm. In *SIGMETRICS*, 56–66.

[73] Poosala, Viswanath, and Yannis E. Ioannidis. 1997. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 486–495.

[74] Quinlan, J. R. 1986. Simplifying decision trees.

[75] Ramamurthy, Ravishankar, and David J. DeWitt. 2005. Buffer-pool aware query optimization. In *CIDR*, 250–261.

[76] Reddy, Naveen, and Jayant R. Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *VLDB*, 1228–1240.

[77] ———. 2005. Analyzing plan diagrams of database query optimizers. In *VLDB*, 1228–1240.

[78] Reiser, M., and S. S. Lavenberg. 1980. Mean-value analysis of closed multichain queuing networks. *J. ACM* 27(2):313–322.

[79] Ross, Sheldon. 2009. *A First Course in Probability*. 8th ed. Prentice Hall.

[80] Rusu, Florin, and Alin Dobra. 2008. Sketches for size of join estimation. *ACM Trans. Database Syst.* 33(3).

[81] Scilab Enterprises. 2012. *Scilab: Free and open source software for numerical computation*. Scilab Enterprises, Orsay, France.

[82] Selinger, Patricia G., Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access path selection in a relational database management system. In *SIGMOD*, 23–34.

[83] Sevcik, Kenneth C. 1981. Data base system performance prediction using an analytical model (invited paper). In *VLDB*, 182–198.

[84] Soror, Ahmed A., Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. 2008. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 953–966.

[85] Stillger, Michael, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's learning optimizer. In *VLDB*, 19–28.

[86] Suri, Rajan, Sushanta Sahu, and Mary Vernon. 2007. Approximate mean value analysis for closed queueing networks with multiple-server stations. In *IERC*.

[87] Tomov, Neven, Euan W. Dempster, M. Howard Williams, Albert Burger, Hamish Taylor, Peter J. B. King, and Phil Broughton. 2004. Analytical response time estimation in parallel relational database systems. *Parallel Computing* 30(2):249–283.

[88] Tozer, Sean, Tim Brecht, and Ashraf Aboulnaga. 2010. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*, 397–408.

[89] Tzoumas, Kostas, Amol Deshpande, and Christian S. Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB* 4(11):852–863.

[90] Unterbrunner, Philipp, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2009. Predictable performance for unpredictable workloads. *PVLDB* 2(1):706–717.

[91] Wasserman, Ted J., Patrick Martin, David B. Skillicorn, and Haider Rizvi. 2004. Developing a characterization of business intelligence workloads for sizing new database systems. In *DOLAP*, 7–13.

[92] Winkelbauer, Andreas. 2012. Moments and absolute moments of the normal distribution. *arXiv preprint arXiv:1209.4340*.

[93] Wu, Wentao, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB* 6(10):925–936.

[94] Wu, Wentao, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, 1081–1092.

[95] Wu, Wentao, Xi Wu, Hakan Hacigümüs, and Jeffrey F. Naughton. 2014. Uncertainty aware query execution time prediction. *PVLDB* 7(14):1857–1868.

[96] Xiong, Pengcheng, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan Hacıgümüş. 2011. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *SOCC*, 15:1–15:14.

[97] Zaitsev, P., and V. Tkachenko. 2012. *High performance mysql: Optimization, backups, and replication*. O'Reilly Media.

[98] Zhang, Ning, Jun'ichi Tatemura, Jignesh M. Patel, and Hakan Hacıgümüş. 2011. Towards cost-effective storage provisioning for DBMSs. *PVLDB* 5(4): 274–285.