Predicting Query Execution Time using Statistical Techniques

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Engineering
IN
Computer Science and Engineering

BY

Priyanka Sharma

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2017

Declaration of Originality

I, **Priyanka Sharma**, with SR No. **04-04-00-10-41-15-1-12852** hereby declare that the material presented in the thesis titled

Predicting Query Execution Time using Statistical Techniques

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2015-2017**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name:Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Lord and beloved Family

Acknowledgements

I am deeply grateful to my mentor Prof. Jayant R. Haritsa for his guidance throughout my project work. It had been a great experience to work under his supervision.

I am also very thankful to Prof. Chiranjib Bhattacharyya and Mrs. Rekha Singhal (Senior Scientist, Tata Consultancy Services) for their valuable suggestions and useful discussions that happened during the project work.

My sincere thanks goes to my fellow lab mates for all the help and suggestions. Also, I am grateful to all the faculty and staff at the Department of Computer Science and Automation for their support and timely help.

Lastly but most importantly, I'll like to thank my parents and my brother for having faith and confidence in me, for encouraging and supporting me.

Abstract

Predicting query execution time is crucial for a number of tasks in database systems such as query scheduling, progress monitoring and costing during query optimization. Query optimizers uses two separate estimation models to find an optimal plan to execute a query a) selectivity estimators to predict the number of input tuples for each operator node of a query plan tree b) cost model to derive execution cost for a given plan. Significant errors occur in estimation of execution time of a query as a result of errors in selectivity estimates as well as inaccuracies in cost modeling [15].

In this work we are trying to minimize the error in the prediction of query execution time which occurs because of inaccurate cost model in the current optimizer of PostgreSQL database. We built a learning based cost model as opposed to traditional analytical models which are predominant in the current optimizer. We chose learning technique over analytical methods because it can capture operator interactions that happen within the query plan. A modeling technique was proposed previously [13] to learn query execution behavior at a fine-grained operator level. Combining this with a different learning technique, we are able to produce significantly better estimates than state of art the technique for a set of (test set) benchmark queries. We evaluate our approach using the TPC-H [4] and TPC-DS [6] workload on PostgreSQL 9.6 [1].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Current PostgreSQL database query optimizer has predictors that rely on some system performance constants and modelling power of the methods available in literature is not enough to obtain them accurately. The PostgreSQL cost model is designed to compare the costs of alternative query plans by estimating the time taken to run a query plan. The time is measured in cost units that are abstract, but conventionally mean disk page fetches [2]. While they do a good job of comparing the costs of alternative plans, they are poor predictors of plan execution time. As a result accurate prediction of execution time remains difficult to achieve for most of the queries. Database systems can have many benefits from accurate estimation of physical execution time under a given hardware and system configuration. It has a wide range of applications including:

- Admission control: Resource managers can use this metric to perform workload allocations such that the specific Quality of Service (QoS) goals are met.
- Query Optimizer: Optimizer can choose best plan for a query based on estimated execution time.
- Query Scheduling: Knowing the execution time is crucial in deadline and latency aware scheduling.
- 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.

1.1 Background

The problem of estimating execution time is well studied over the past decade. Figure 1.1 illustrates the classical architecture of a query optimizer. Given a query, the job of query optimizer is to look at the various enumerations of query plan and pick the cheapest one. Each query plan is assigned a cost based on the operators, choice of implementation and input cardinalities and this cost is estimated by the cost model. Ideally we prefer the estimated cost to be close to the actual time taken to run the query, but in practice, this is rarely the case primarily because of the errors from the following components.

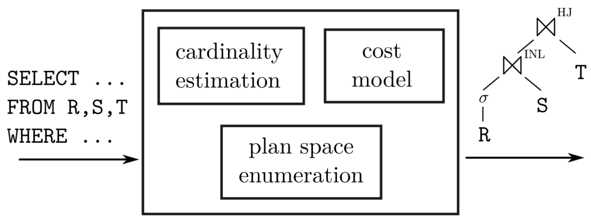

Figure 1.1: Query Optimizer Architecture

- Cardinality estimators: It is well studied that predicate selectivity estimates used for optimizing SQL queries are often significantly in error [9].
- Cost model: The job of the cost model is to look at the query plan with estimated cardinalities and come up with an estimate of the execution time. Current query optimizers predominantly use analytical cost formulas to compute cost based on the amount of data flowing through the operators. Cost model needs tuning if they are to produce estimates specific to a target hardware. However a tuned formula often cannot compensate for the effects introduced by complex query optimizations (e.g., intra-query parallelism).

These reasons are studied in previous works and they have found the current optimizer constants can't be used directly for prediction of the query execution time. For example, in [8], the authors found that using linear regression to map execution time to cost of Neoview's commercial query optimizer was not effective (Figure 17 of [8]). Once more similar results were obtained in [7] which tried to map execution time to PostgreSQL optimizer cost estimates using same technique (Figure 5 of [7]). The reason for this is there exists a non-linear relationship between estimated cost using current optimizers and the actual running time of the query [10].

The error metric plays an important role to measure the accuracy of any work. The error metric used by previous work (Relative Error) is bounded as well as biased towards underestimations. Therefore, we are using a different metric called Q-error introduced in [11] as compared to all of the previous work for the calculation of the error in the prediction of query execution time on a PostgreSQL database. Other details of this metric are discussed in the Section 4.1 [9].

1.2 Our Contribution

- We propose statistical techniques which learn query behavior at a finer granularity i.e., at physical implementation level of a query operator. This operator level learning was first introduced in [7]. They are using linear regression for the model building but in contrast we are using support vector regression (SVR) and Gradient Boosted regression trees (GBRT) which are able to fit non-linear dependencies found in the training data. We have shown these non-linear approaches out-performs the linear approach [17] in Section 4. We then compose these individual operator running time estimates to produce an overall execution time estimation for a given query plan.

- Results on benchmark workloads suggests that the proposed technique outperforms the state of the art.

1.3 Roadmap

The rest of the report is organized as follows: we start with the related work and applicability of their solutions to the current problem in Chapter 2. An overview of proposed approach in discussed in Chapter 3. We introduce operator level models and the feature selection process in Sections 3.3 and 3.4 of Chapter 3, respectively. We then describe the training and testing phases in Sections 3.5 and 3.6 of Chapter 3, respectively. We validate our approach and present the experimental results in Chapter 4. Next, we discuss the shortcomings of our approach in Section 4.3 of Chapter 4. We conclude the thesis by discussing future work in Chapter 5.

Chapter 2

Related Work

Recent work has explored the use of machine-learning based techniques for the estimation of both run-times as well as resource usage of SQL queries, both for queries in isolation [7, 10], as well as in the context of interactions between concurrently executing queries [16].
The work done in [8] is the first of its kind to embed machine learning techniques inside query optimizer. They are doing a Plan-level modeling in which each query plan has a feature vector associated with it. This feature vector is used for training the model as well as for the estimation of a new query. In their approach, two queries are said to be similar if their plan have same or similar feature vector. Their approach has some limitations. As resource estimate for a query Q is obtained by averaging the resource characteristics of the three queries in the training data that are the most similar to Q, this technique is not capable of "extrapolating" beyond the training data [10].

For static workloads, where the incoming queries are simply instances of an already known query template this approach is suitable but it lacks the generalizing ability required for ad-hoc queries.

The approach proposed in [7] mitigates the issue mentioned above to some extent by introducing operator level models and using it along with the plan level models for predicting execution time. The models offer the generalization properties to an extent but their ability is limited by the choice of machine learning method they have used for operator level modeling [10]. The authors use linear regression models for each operator; that means they implicitly force the output to vary linearly with each input feature. However we have shown in our experiment section the execution time of a query doesn't vary linearly with the features.

The work of [10] explores the use boosted regression trees along with scaled functions. They primarily focus on estimating the logical CPU and I/O consumption for a query. In contrast, we focus on estimating the actual execution time (physical). Moreover, we are taking care of intra-query parallelism by choosing our feature set accordingly whereas their work does not support it.

The approach proposed in [17] looks at tuning the internal cost parameters of PostgreSQL engine. They do this by running a set of calibrated queries and computing the values of cost parameters. Their approach can generalize to ad-hoc queries and produce estimates with some accuracy making it the state of the art. However being analytical they lack the power to account for query interaction among operators (Compute & I/O) overlap. In contrast, we try to account for these effects produced by those subtle optimizations by looking at patterns on actual query executions on a target hardware.

As of now we have implemented the state of the art [17] approach. We have some experimental evidence which shows our approach outperforms the approach proposed in [17].

Chapter 3

Proposed Approach

3.1 Assumptions

Before we go into the details of the approach, we list the constraints we impose to solve the problem.

- We only consider predictions for standalone queries.
- Perfect selectivity estimates are assumed. This allows us to study the sole impact of cost model on estimation errors.
- System configuration remains the same through-out the whole process.
- Number of children at any node in query tree are limited to 2. Most operators in the engine are unary or binary. However there are tiny fraction of operators which can have more than 2 children (e.g., Bit-OR, Bit-AND), here the number of children are unbounded.

3.2 Overview

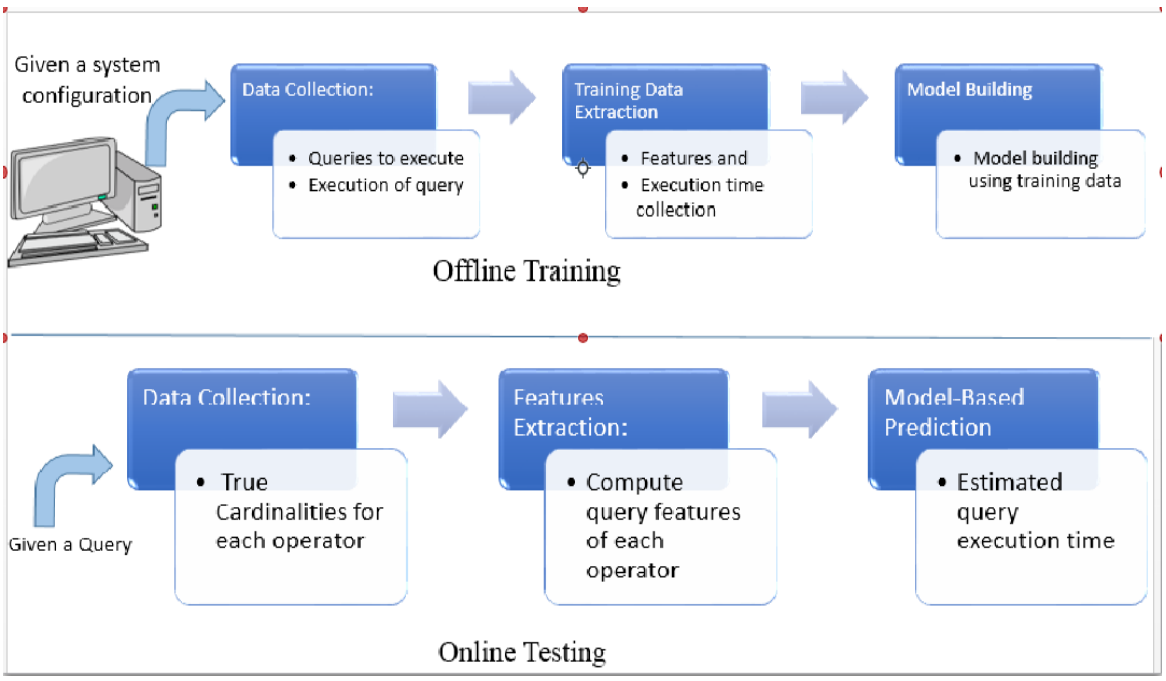

Figure 3.1: Overview of proposed approach

Our approach consists of two distinct phases: an off-line Training and an on-line Testing. During training phase, we execute a set of queries on the target system and collect the values of features for each operator. For each physical implementation of an operator, we build a model to predict the response time (time taken to produce all the tuples) taken by that operator. We recursively use information from child operators to produce an estimate for its parent operator. Hence, the total execution time of a query will be response time of the root node in the query plan tree. Since there are only handful of operators in the engine, this approach of learning models for each operator is feasible in terms of both time and space complexity.

3.3 Operator Level Models

We are trying to predict the execution time of arbitrary queries not seen during training. For this purpose, we use the fact that query plan is composed of a set of SQL operators within database engine already. As we are building model for each operator and because this approach mimics the way SQL queries are executed, this allows us to make predictions for any arbitrary query by composing individual operator estimates. Training an operator involves extracting individual running time of an operator from the total execution time. For this purpose we are using PostgreSQL *explain analyze* command, given a query, it gives the individual operator response time for all the operators involved in the query. We use these actual response time to build a learning model for predicting execution time for each operator.

3.4 Features

Feature selection process deals with issue of choosing most predictive feature set. For each operator we have used different sets of features to create models according to their working mechanisms. Feature selection is based on domain knowledge of database internals and working of individual operators. Some of them were also used by previous work [8] [7] [10].
In PostgreSQL 9.6 (current version) they have introduced a facility to execute some operators in parallel i.e multiple processes will be allocated to a particular portion of a plan and every background process which is successfully started for a given parallel query will execute its portion of the plan. We are including the effect of this parallelism by introducing some additional features in our model.
As previously mentioned we have used two learning techniques SVM and GBRT for modeling the execution time of the operators. The model building process using GBRT facilitates us to see the order of importance of features used for training the model for an operator. The relative rank (i.e. depth) of a feature used as a decision node in a tree can be used to assess the relative importance of that feature with respect to the predictability of the target variable. Features used at the top of the tree contribute to the final prediction decision of a larger fraction of the input samples. The expected fraction of the samples they contribute to can thus be used as an estimate of the relative importance of the features. By averaging those expected activity rates over several randomized trees one can reduce the variance of such an estimate and use it for feature selection. In python implementation of those estimates are stored as an attribute named feature_importances_ on the fitted model. This is an array with shape (n_features,) whose values are positive and sum to 1.0 [5]. The higher the value, the more important is the contribution of the matching feature to the prediction function.
Values for features are extracted from PostgreSQL 9.6 *explain analyze* command.

Feature Description
Estimated Number of input tuples from child
Estimated Average width of input tuples from child
Estimated Average width of output tuples
Estimated sequential pages fetched
Estimated random pages fetched
Estimated tuples accessed via index
Estimated number of CPU operations to be performed
Estimated number of tuples transferred from parallel worker process
Workers Launched

Table 3.1: List of features for Sequential Scan operator.

Table 3.1 contains the list of features used for the Sequential Scan operator. The feature importance graph i.e. Figure 3.3, shows that sequential pages scanned and CPU operations performed are the two most related features with the time needed to execute sequential scan when it is executed sequentially, which is true as one would have expected theoretically. Increase in sequential pages scanned will increase the time of sequential scan as well as increasing filter constraints will result in more CPU operations which eventually increase the time needed to perform sequential scan.
Here, the features “Number of input tuples from LC ” and “Estimated sequential pages scanned” seems to be dependent. However because of the operator pipelining during query execution they are not totally dependent. To understand this concept consider query plans given in Figure 3.2.

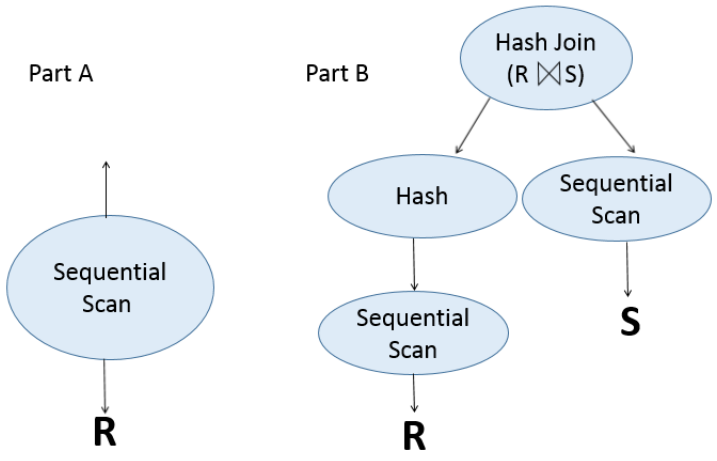

Figure 3.2: Example Query Plans

In the part A, there is only a sequential scan on relation R. So for this plan the "sequential pages scanned" feature contains the number of pages fetched that contains the R data. Whereas in part B there is a hash operator above the same sequential scan operator. In this case both the sequential scan operator and hash operator will execute together (operator pipelining) and hash bucket will be generated for all different values of the join attribute. Sometimes working memory allocated for the query execution gets full before generating complete hash table. Then to make space for new hash bucket operating system removes the older one and when the removed bucket is required it is fetched in the main memory. Because of this some addition will be there in the "estimated sequential pages scanned" feature. In PostgreSQL, as both the operators run simultaneously, the combined effect is reflected in both operators feature i.e the value of "estimated sequential pages scanned" feature for both will be same and their response time will also be almost same. So, for same number of input tuples the feature sequential scanned pages will have different value. Hence, in our case these two features are not directly dependent and we need to have both in our feature set.
When the sequential scan executed in parallel, number of process that are launched for its execution, should be a factor on which its execution time depends. Our model captures this fact and can be seen in Figure 3.4.

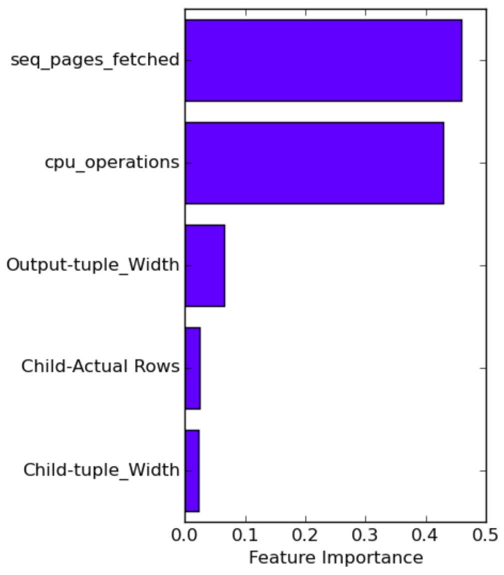

Figure 3.3: Feature importance for Sequential Scan operator when executed sequentially

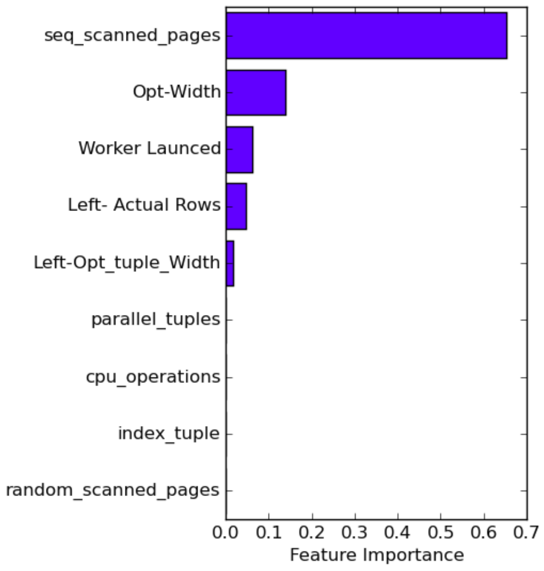

Figure 3.4: Feature importance for Sequential Scan operator when executed in parallel

Feature Description
Estimated Number of input tuples from child
Estimated Average width of input tuples from child
Estimated Number of output tuples
Estimated Average width of output tuples
Estimated random pages fetched
Estimated tuples accessed via index
Estimated number of CPU operations to be performed
Actual Loops

Table 3.2: List of features for Index Scan operator.

Table 3.2 shows the feature set used for index scan operator. As parallelism is not supported by index scan operator in the current implementation of database it doesn't have any feature related to it.

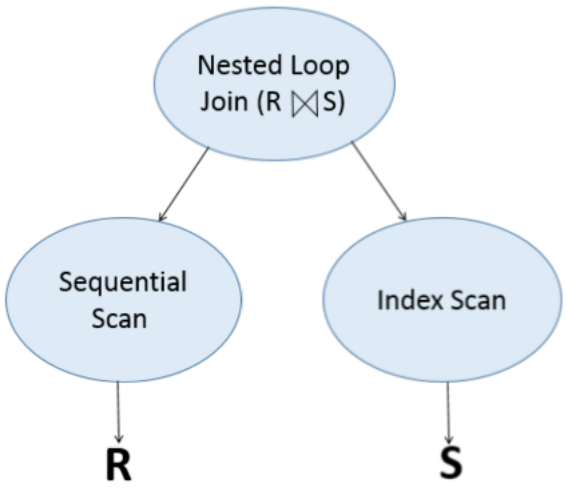

Figure 3.5: Example Query plan tree

All above mentioned features have very transparent meaning except the Actual Loops. In some query plans, it is possible for a subplan node to be executed more than once. For example, consider a plan shown in Figure 3.5, joining two relation using nested loop join. It uses sequential scan and index scan on the outer and inner relation respectively. So, the index scan operator will be executed once per outer row (output of sequential scan operator) in the plan. In such cases, the Actual Loops value in the output given by *explain analyze* command of

PostgreSQL reports the total number of executions of the inner node, and the execution time and rows (Estimated number of output tuples) given are the average over number of iterations (Actual Loops) of execution time and output rows respectively [3]. So, we multiply execution time and row count of that operator by the Actual Loops to get the total time actually spent in the node and the exact number of the output tuple from that node respectively.
The feature importance graph for index scan is shown in Figure 3.6.

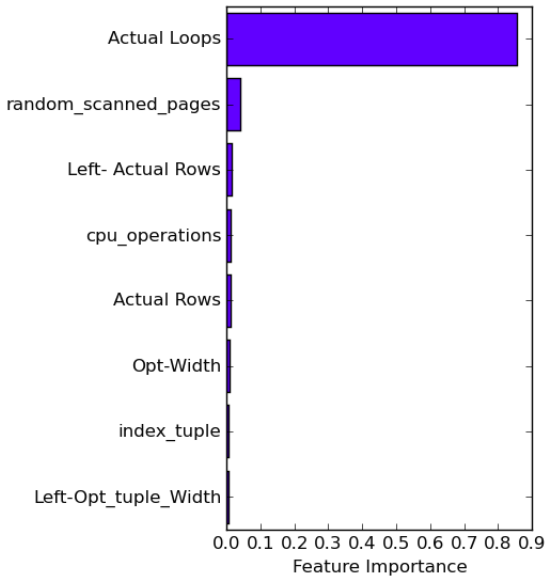

Figure 3.6: Feature importance for Index Scan operator

.

Feature Description
Estimated Number of input tuples from child
Estimated Average width of input tuples from child
Estimated Average width of output tuples
Estimated sequential pages fetched
Estimated random pages fetched
Estimated tuples accessed via index
Estimated number of CPU operations to be performed
Estimated number of tuples transferred from parallel worker process
Workers Launched

Table 3.3: Features used for Aggregate and Hash operators.

Table 3.3 contains set of features used for Aggregate and Hash operators. Feature importance graph for them is shown in Figures 3.7 and 3.8 respectively, where we can see that the most important feature for both the operators is estimated number of rows coming from child operator which is very obvious because more the number of input tuples are there more time is required by the operator to process them.

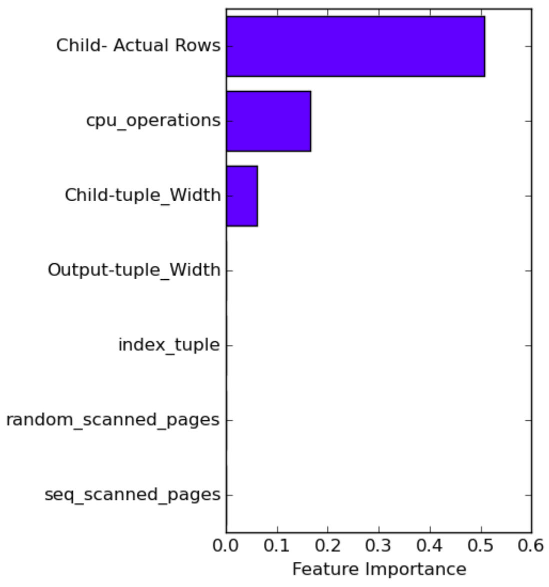

Figure 3.7: Feature importance for Aggregate operator

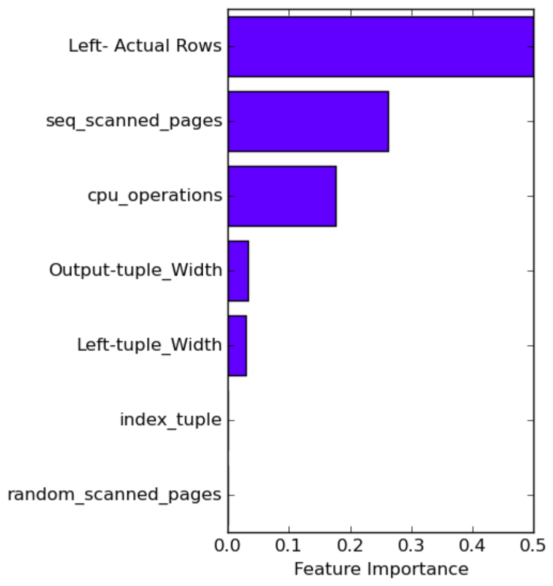

Figure 3.8: Feature importance for Hash operator

Feature Description
Estimated Number of input tuples from child
Estimated Average width of input tuples from child
Estimated Average width of output tuples
Estimated sequential pages fetched
Estimated random pages fetched
Estimated tuples accessed via index
Estimated number of CPU operations to be performed
Sort key: how many attributes are there in order by clause

Table 3.4: List of features for Sort operators.

Table 3.4 features are used for Sort operator and feature importance for sort operator quicksort is shown in Figure 3.9, which shows sorting time is highly dependent on the number of tuples coming from child operator to sort which is obviously true.

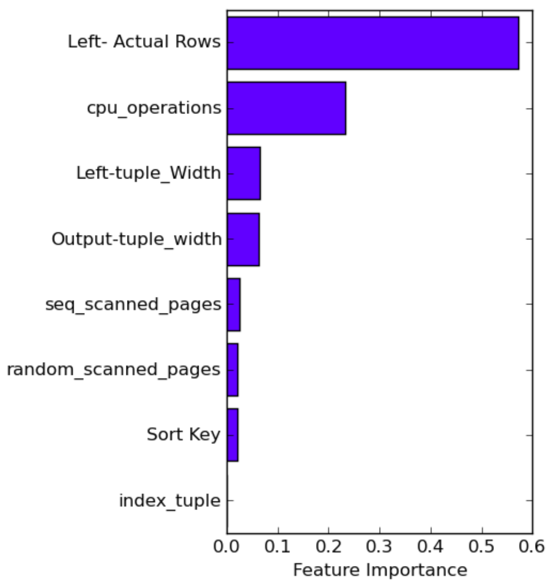

Figure 3.9: Feature importance for Quick Sort operator.

Feature Description
Estimated number of input tuples from LC
Estimated Average width of input tuple from LC
Estimated number of input tuples from RC
Estimated Average width of input tuples from RC
Estimated Average width of output tuples
Estimated sequential pages fetched
Estimated random pages fetched
Estimated tuples accessed via index
Estimated number of CPU operations to be performed
Estimated number of tuples transferred from parallel worker process
Workers Launched

Table 3.5: List of features for Join operators. LC and RC indicates left child and right child operator respectively.

Table 3.5 features are used for all the join operators and feature importance for hash join operator is shown in Figure 3.10 and it again shows that the most important feature for operator is the number of tuples coming from the child node. Here it shows that the hash join operator is more dependent on the number of rows coming from left child rather then number of tuples coming from right child because “Hash Join node” in PostgreSQL query plan has scan operator as its left child and hash operator as its right child. The number of tuple coming out of the scan operator will get probed into hash table to check whether it satisfies the joining condition or not. Therefore, if the selectivity of the filter in the scan is high, hash join operator will take more time to complete the join.

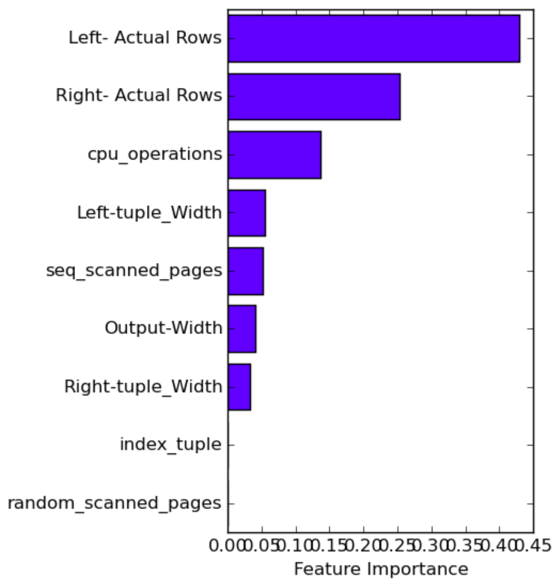

Figure 3.10: Feature importance for Hash Join operator.

The operator having features estimated number of tuples transferred from parallel worker process and workers launched will be present in the features set of that operator only when the operator is getting executed in parallel.

The features number of output tuples, sequential pages scanned, random pages scanned, tuples accessed via index, number of CPU operations to be performed and number of tuples transferred from parallel worker process is also used by current PostgreSQL database optimizer for calculating cost (an abstract unit) of the query execution by multiplying each one with the cost associated with the respective task and then adding them to get the final cost for the query execution. As the cost with which they are getting multiplied are abstract constant whose values are inaccurate, therefore, cost required for the query execution doesn’t represent the time needed for its execution. As the values of these features were not directly available in the *explain* command of PostgreSQL, we have added a functionality in command, now it gives the values for these features at the compile time of the query.

Having described about selection of these features, we now move on to explain how the predictions for a query plan can be made. Query plan predictions are computed by composing the individual operator predictions; specifically we traverse query tree in a post-order fashion (left child, right child and root). The parent operators use the estimates produced by their children as part of the features. This allows us to build predictions progressively. On the downside this also means that prediction errors are propagated.

3.5 Training

In this phase, prediction models are derived by executing set of queries and observing their true running times. More precisely, example instances of execution time are collected for each operator present in query plan. By executing more number of queries, we can obtain more example instances which can in turn lead to better predictions. The training queries need to be chosen such that:

- The example instances need to be significantly “different”. By difference between training examples we mean that the variance within each feature is large. This can be achieved to an extent by executing queries under a combination of different scales and data distribution.
- They cover all the operators. This is particularly important because it directly determines what type of testing queries are allowed. We cannot have a query plan which contains an operator that is not associated with prediction model. Usually, Query Optimizer is sensitive enough to produce different plans with change in data distribution and size.
- Each operator has sufficient number of example instances. Some operators are sparse in nature and they are not readily available in query plan. To handle this, we need to add hand-tuned queries that contain a specific operator. Running these queries under different tables, scales will mitigate the issue to some extent.

Once we have the example instances, we now need to find the prediction model that best explains the given data. First we tried Linear regression model for the prediction but the results we got was not good, this is because of their inability to model non-linear relations among features and the execution time [10]. Therefore we tried working with Support Vector Regression (SVR) using Radial Basis Function (RBF) as its kernel and Gradient Boosted Regression Trees

(GBRT).
SVR need to be tuned properly to fit the underlying data [13]. There are two parameters associated with it:

- C - This is a regularization constant that trades complexity for accuracy. Low values usually produce very simple models that underfit, while large values tend to overfit the data.

- γ - This is a kernel coefficient for RBF. Intuitively, the gamma parameter defines how far the influence of a single training example reaches. If gamma is too large, the radius of the area of influence of the support vectors only includes the support vector itself and no amount of regularization with C will be able to prevent overfitting. When gamma is very small, the model is too constrained and cannot capture the complexity or “shape” of the data.

C, γ are continuous variables and finding ideal values require an exhaustive search. Trying exponentially growing sequences of C and γ is a practical method to identify good parameters.

$$C = \{2^{-3}, 2^{-1}, \ldots, 2^{15}\},$$
$$\gamma = \{2^{-10}, 2^{-8}, \ldots, 2^{3}\}.$$

The above range for C and γ is what is usually recommended in literature for fitting complex functions. Note that SVR also requires the individual features to be normalized which implies that we need to have the same scaling functions for both training and testing. Hence, we also write the corresponding scaling function to disk

The goal of GBRT is to combine the predictions of several base decision trees in order to improve robustness over a single estimator. We chose to try GBRT after SVM for our case because these models are able to arbitrarily break the domain of input features and does not rely on a fixed functional form. Therefore, is able to fit very complex non-linear relationship between features and target. We trained GBRT using 1k boosting stages for each model where each decision tree can have at most 10 leaf nodes.

In this way, for every operator we learn a prediction model and materialize them to disk. . The overall space consumption (including models and scaling functions) is around 36MB.

3.6 Testing

For a given query plan, we predict the execution time by traversing through its nodes in post-order fashion. At each node, we invoke the earlier materialized model to get an estimate. The children data is then used by parents to produce their own estimate. The estimated execution time of a given query plan is therefore estimated execution time of *root* node.

Chapter 4

Experimental Evaluation

In this section we evaluate the accuracy and robustness of our technique along with the state of the art [17]. To evaluate the robustness we have specifically taken test queries which are different from that of training.

4.1 Setup

- Database management system: PostgreSQL 9.6 [1].
- Datasets and Workload: We have used TPC-H and TPC-DS benchmark [4] [6] queries for training the predictive models. For TPC-H, we generated underlying data distribution by a tool published at [12]; this generates data which follows Zipfian distribution and allows us to control the degree of skewness by setting a value for Z (with 1 being uniform and 4 being highly skewed). In our experiments we have set Z to 2 which ensures that there are significant differences between queries even among the same template. The TPC-DS database also assumes skewed distribution of data.
  We have used QGEN [4] tool to generate 590 queries; having mixture of queries from total of 21 query templates of TPC-H on 5GB database. We have generated 90 queries from templates of TPC-DS using DSQGEN [6] tool on 5GB database. For testing, we have separately generated one query for each TPC-H template and 7 queries from TPC-DS template and checked on 5GB database. The query templates used for generating TPC-DS test queries are different from the templates used for generating training queries, whereas in case of TPC-H, test queries have same template but queries are different from the queries in training set.
- Hardware: All the training and testing queries were executed on a machine with 3.0 Ghz Intel Core Extreme processor and 8 GB of RAM. Queries were executed sequentially by

flushing operating system cache and database system buffers.

- Predictive models: We have used Python's sckit-learn [?] library for all the ML implementations.

- Error Metric: We have used the Q-Error (QE) [9, 11] as our error metric.

  $$\text{QE}(a,b) = \text{Max}(\frac{a}{b}, \frac{b}{a})$$

  Similarly, we use Average Q-Error(AQE) to compare the efficiency for a set of queries. This metric is useful when we would like to minimize the prediction error regardless of their execution time. Other metrics like square error are useful when we want to minimize the absolute difference between actual and predicted time. In previous work [17, 7, 10], authors have used metric called Relative Error(RE). It is defined as:

  $$\text{RE}(a,b) = \frac{|a-b|}{a}$$

  The problem with this metric is that it is biased towards under estimation i.e., we can always underestimate the value of b (e.g., 0) and get a RE of 1.0. As such, in many cases they can have deceptively low Mean RE even though the actual estimates have high error. In contrast, Q-Error metric is unbounded and catches under estimations.

- Alternative techniques: We compare the accuracy and generalization ability of approach proposed in [17]. As in their case, we will assume perfect selectivity estimates while producing the results. For ease of reference, we refer their [17] approach as Tuning and our approach as Learning.

4.2 Evaluation

4.2.1 Non-Linear Relation Between Execution time and Features

As we have claimed that the execution time of a query is not linearly dependent on the feature set of an operator. In this section we are trying to analyze what kind of non-linear relationship exists between execution time and the features of an operator. As we have seen in Section 3.4, the minimum number of feature in any operator is 8. As one can't visualize 9 dimensional regression curve and only few features was having impact on the execution time of the operator, we are studying the impact of two most important features on the operator.

We took instances of sequential scan operator (when executed sequentially) for this analysis. According to our model the two most important feature in this case are “Estimated sequential scanned pages” and “Estimated CPU operations”. We plotted a scatter graph between these two features and the execution time of the operator. Figure 4.1 shows the plot where in Part A the z-axis represents actual execution time taken from training data and in Part B z-axis represents predicted execution time of the operator which is predicted using our model.

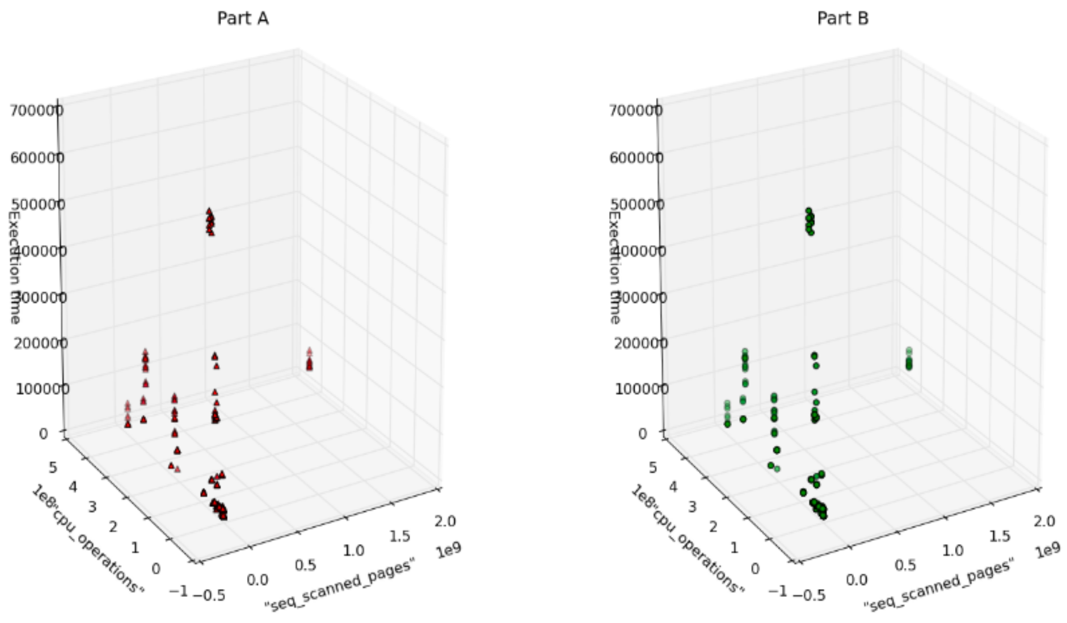

Figure 4.1: Scatter Plot between Estimated Sequential Scanned pages, Estimated CPU operations and the execution time for a Sequential Scan operator (when executed sequentially).

We observed in x-axis near 2×10^8 value a small interval is there in which value of *sequential_scanned_pages* feature is almost constant, So we took that interval to study the impact of *cpu_operations* on the sequential scan operator. When we plotted the values of *cpu_operations* for that interval against the execution time of sequential scan operator we got the regression curve shown in the Figure 4.2. We are getting a step function because we are using a technique which are nothing but combination of many small decision trees and we know regression using decision trees generates a step like function.

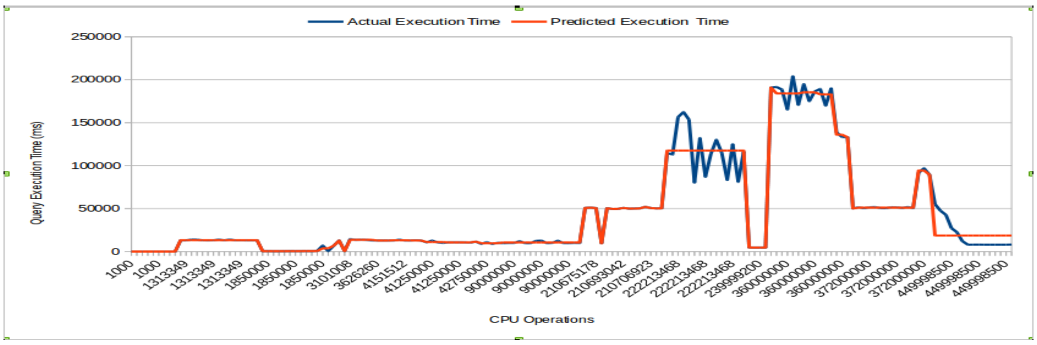

Figure 4.2: Non-linear relation between Estimated CPU operations and Execution time for a Sequential Scan operator (when executed sequentially).

4.2.2 Training Data Size: 5GB Test Data Size : 5GB

In this experiment we have generated the test queries using QGEN and DSQGEN tools. We made sure that the training and testing queries do not contain identical query (i.e., same template and instance). First we have shown how linear regression performed for the prediction of query execution time as compared to GBRT in Table 4.1. Sometimes it performed better than GBRT but when it goes bad it can go worse upto many orders of magnitude.

Query	QE(Linear)	QE(GBRT)
1	1.5 (-)	2 (-)
4	1.9 (-)	1.2 (-)
5	1.8 (+)	1.1 (-)
7	20.2 (-)	1.3 (-)
8	9.2 (+)	1
9	3.7 (+)	1
10	1.1 (+)	1
17	4.2 (-)	1
19	3.4 (+)	1.1 (+)

Table 4.1: Per Query Q-Error(QE) for TPC-H (5-GB), (-): Underestimate (+): Overestimate.

We have shown the per-query error in Table 4.2 for TPC-H queries on 5-GB database and in Table 4.3 for TPC-DS queries on 5-GB database. We can see in Figure 4.3 and 4.4 that GBRT performs better as compared to other techniques here.

Query	QE(Tuning)	QE(SVM)	QE(GBRT)
1	16.3 (-)	1.8 (-)	2.0 (-)
4	16.5 (+)	2.6 (-)	1.2 (-)
6	15.6 (-)	1.1 (-)	1.1 (-)
7	17.7 (-)	1.4 (-)	1.3 (-)
8	9.8 (-)	1.1 (-)	1
10	9 (-)	1	1
11	2.6 (+)	1.4 (+)	1
12	11.3 (-)	1.4 (-)	1
13	25.1 (+)	1	1
18	9.7 (-)	1	1
19	2.3 (+)	1	1.1 (+)

Table 4.2: Per Query Q-Error(QE) for TPC-H (5-GB). (-): Underestimate (+): Overestimate

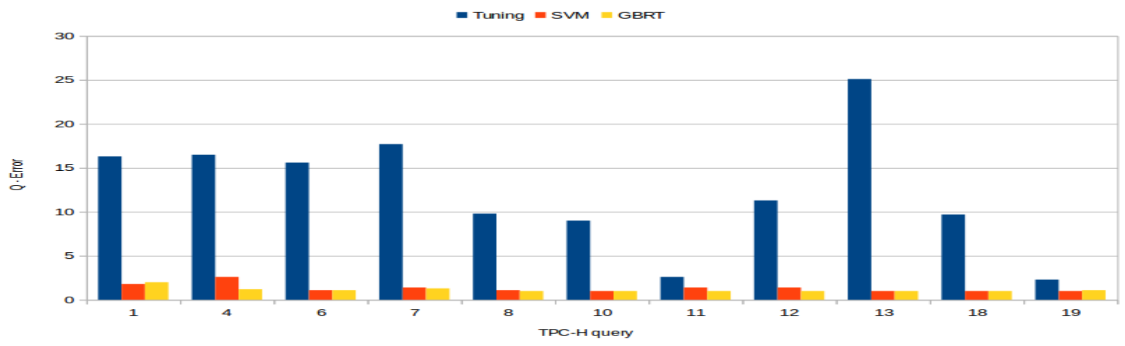

Figure 4.3: Comparison between Tuning and Learning for TPC-H (5-GB)

Query	QE(Tuning)	QE(SVM)	QE(GBRT)
1	9 (-)	2 (-)	1.6 (-)
2	1.2 (-)	4.3 (-)	3.7 (-)
3	12.3 (-)	1	2.7 (-)
5	7.7 (-)	2.4 (+)	2.6 (+)
6	8.9 (-)	2.3 (+)	3.2 (+)
7	2.3 (+)	2.1 (-)	1.9 (+)
10	9 (-)	3.1 (+)	1.3 (-)

Table 4.3: Per Query Q-Error(QE) for TPC-DS (5-GB). (-): Underestimate (+): Overestimate

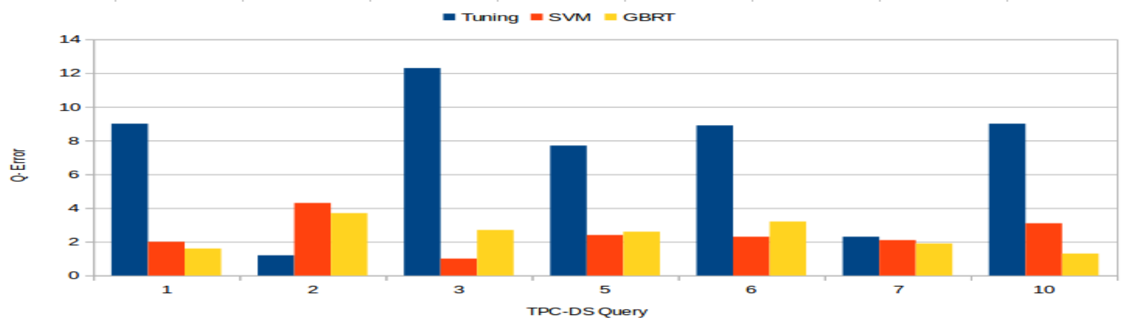

Figure 4.4: Comparison between Tuning and Learning for TPC-DS (5-GB)

4.2.3 Training data size : 5GB and Test data size: 10GB

Next we ran same test queries using same models learned before but on a larger scale database (10GB) and found that the error got increased and most of the predictions was underestimates. As we are using instance from 5GB and testing on 10GB, the model learned is not able to scale up properly with the increase in the data size. Error for each TPC-H query for this scenario is shown in Table 4.4.

Operator	QE(Tuning)	QE(SVM)	QE(GBRT)
2	37.9 (+)	1	1.9 (-)
4	26.2 (-)	6.8 (-)	3.7 (-)
5	17 (-)	2.6 (-)	2.6 (-)
6	15.3 (-)	1.9 (-)	1.8 (-)
7	19.9 (-)	3.5 (-)	2.9 (-)
8	11.1 (-)	2.1 (-)	2.1 (-)
10	9.4 (-)	1.9 (-)	1.5 (-)
11	3.1 (+)	4.7 (+)	1.7 (-)
14	16.5 (-)	2 (-)	2.1 (-)
17	14.6 (-)	3 (-)	1.7 (-)
18	11.1 (-)	1.9 (-)	1.8 (-)
19	1.5 (-)	1.4 (-)	1.6 (-)

Table 4.4: Per Query Q-Error(QE) for TPC-H (Training using 5GB and Testing using 10 GB database). (-): Underestimate (+): Overestimate

4.2.4 Results Summary

As we are doing operator level modeling, analyzing error generated at the operator level is very important. Table 4.5 contains the individual operators errors which we got after training on TPC-H 5GB database.

Query	QE(Min)	QE(Avg)	QE(Max)
Seq Scan (sequential)	1	1.2	2.1
Seq Scan (parallel)	1	1.6	4.2
Index Scan	1	2.3	1.8
Aggregate (Sorted)	1	1.2	2
Aggregate (Hashed)	1	1.4	2
Aggregate (Plain)	1	1.1	1.9
Hash	1	1.3	3
Hash Join	1	1.4	3.5
Sort - Merge	1	1.2	1.6
Sort - Quick	1	1.1	1.4
Nested Loop	1	2.7	4.2

Table 4.5: Per Operator Q-Error(QE) for TPC-H (Training using 5GB and Testing using 5 GB database).

In Table 4.6, we show the summary statistics for dataset TPC-H (5-GB) and dataset TPC-DS (5-GB). In Table 4.7, we show the summary statistics for dataset TPC-H (10-GB). The average Q-error for 5GB is 1.3 and for 10GB is 2.1.

	Tuning	SVM	GBRT
Average	10.3	1.6	1.3
Minimum	1.2	1	1
Maximum	25.6	4.3	3.7

Table 4.6: Learning and Tuning comparison w.r.t QE, (TPC-H and TPC-DS 5-GB)

	Tuning	SVM	GBRT
Average	15.2	2.7	2.1
Minimum	1.3	1	1.3
Maximum	44.6	5.9	3.7

Table 4.7: Learning and Tuning comparison w.r.t QE, TPC-H (10-GB)

4.3 Limitation

Having discussed the approach, in this section we comment on limitations of our approach:

- With the increase in database size the prediction error is also increasing. That is, our model works well only when training and test queries are from same size database.

Chapter 5

Conclusions and Future Work

In this work, we studied the effect of cost model on overall execution time estimation. By modeling at the level of physical implementation of an operator, we have shown that predictions for ad-hoc queries can be achieved upto an extent.
We believe this work opens up lot of opportunities to further improve the estimates. The feature selection can be improved further to give better predictions. We can use all the different plans possible for a single query for training, as they all will have some different operator used in them. So with the use of single query we will be able to train more number of operator. We can use the Picasso tool [14] for having all possible plans of a query.
We need a portability technique such that we can learn on smaller database and use it on larger one.

Bibliography

[1] PostgreSQL 9.6 DBMS.
https://www.postgresql.org/docs/9.6/static/index.html, 2017. [Online; accessed 27-June-2017]. ii, 21

[2] PostgreSQL Cost model.
https://www.postgresql.org/docs/9.6/static/sql-explain.html, 2017. [Online; accessed 27-June-2017]. 1

[3] PostgreSQL 9.6 Performance Tips.
https://www.postgresql.org/docs/current/static/using-explain.html, 2017. [Online; accessed 27-June-2017]. 13

[4] TPC-H 2.4.0-benchmark specification.
http://www.tpc.org/tpch/, 2017. [Online; accessed 27-June-2017]. ii, 21

[5] Feature importance evaluation.
http://scikit-learn.org/stable/modules/ensemble.html, 2017. [Online; accessed 27-June-2017]. 8

[6] TPC-DS 2.4.0-benchmark specification.
http://www.tpc.org/tpcds/, 2017. [Online; accessed 27-June-2017]. ii, 21

[7] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE)*, pages 390–401, 2012. 2, 3, 4, 8, 22

[8] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering (ICDE)*, pages 592–603, 2009. 2, 4, 8

[9] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *VLDB*, 9(3):204–215, 2015. 2, 22

[10] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *VLDB*, 5(11): 1555–1566, 2012. 2, 4, 8, 18, 22

[11] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *VLDB*, 2(1):982–993, 2009. 2, 22

[12] V. Narasayya. Program for TPC-H Data Generation with Skew. `http://research.microsoft.com/en-us/downloads/98710c69-8db6-4a59-a217-6651c8aabbf4` [Online; accessed 27-June-2017]. 21

[13] Vamshi Pasunuru. Predicting query execution time using statistical techniques. `http://dsl.cds.iisc.ac.in/publications/thesis/vamshi.pdf`, M.E Thesis 2016. ii, 19

[14] Naveen Reddy and Jayant R Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, pages 1228–1239, 2005. 30

[15] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001. ii

[16] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *VLDB*, 6(10):925–936, 2013. 4

[17] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Data Engineering (ICDE)*, pages 1081–1092, 2013. 3, 5, 21, 22