Predicting Query Execution Time using Statistical Techniques

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFIMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Engineering
IN
Faculty of Engineering

BY

Vamshi Pasunuru

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

June, 2016

Declaration of Originality

I, **Vamshi**, with SR No. **04-04-00-10-41-14-1-11163** 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 **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2014-16**.
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 discusions.
- 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: Jayant R. Haritsa Advisor Signature

DEDICATED TO

My parents

For their unconditional love.

Acknowledgements

I would like to thank my mentor Prof. Jayant R. Haritsa for his continuous support, encouragement and freedom to pursue the problem in the way I like. I also thank Prof. Chiranjib Bhattacharyya for his valuable suggestions during various discussions.

Abstract

The ability to estimate the 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 b) cost model to derive execution cost for a given plan. Significant errors occur in estimation of execution time [14] as a result of errors in selectivity estimates as well as inaccuracies in cost modeling.

In this work we study the effect of cost model on predicting execution time by designing a learning based cost model as opposed to traditional analytical models which are predominant in the current query optimizers. Learning based models enable the system to capture the effects of underlying hardware (such as speed of CPU, disk etc.) as well as operator interactions that happen within the query plan. We propose a modeling technique to a) learn query execution behavior at a fine-grained operator level b) capture interactions among operators by using a pipeline aware feature set. Combining this with a powerful learning technique, we are able to produce significantly better estimates for a set of benchmark queries. We evaluate our approach using the TPC-H [3] workload on PostgreSQL [1].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Database systems can greatly benefit from accurate estimation of 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 among alternative plans 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 that take an unreasonably long time to execute.

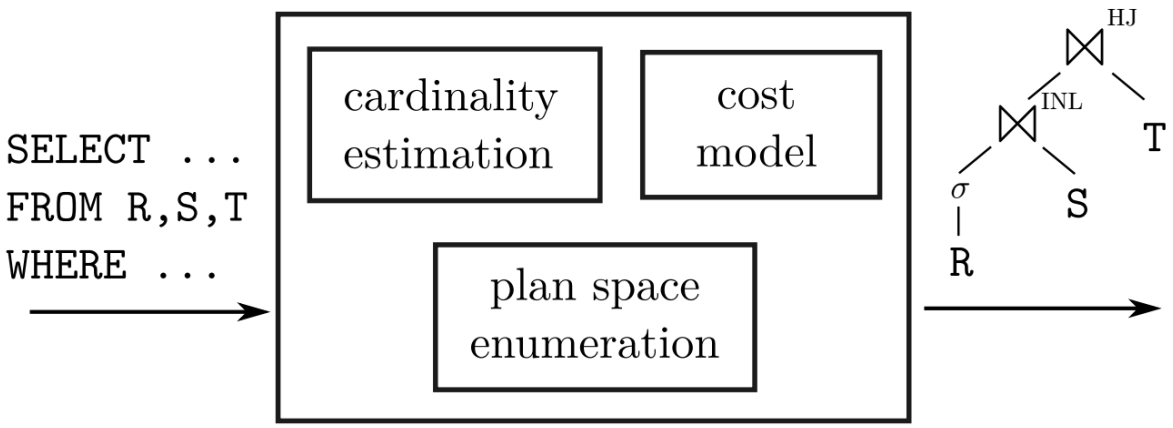

Figure 1.1: Query Optimizer Architecture

The problem of estimating execution time is well studied over the past few years. Figure 1.1 illustrates the classical cost based architecture. 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. Ideally we prefer the estimated cost to be close to the actual time taken to run the query. In practice, this is rarely the case primarily because of the errors from the following components.

- Cardinality estimators: They derive estimates either through meta-data or random sampling. It is well studied that predicate selectivity estimates used for optimizing SQL queries are often significantly in error [8]. The reasons for such substantial deviations are well documented [14], and include outdated statistics, coarse summaries, attribute-value independence (AVI) assumptions and complex user-defined predicates. Often the real life data is skewed and attributes are correlated to each other; resulting in significant estimation errors.

- 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 predominately 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 hand tuned formulas often cannot compensate for the effects introduced by complex query optimizations (e.g., Pipelining).

The PostgreSQL cost model is designed to compare the costs of alternative query plans by estimating the time taken to run a query plan (time measured in cost units that are arbitrary, but conventionally mean disk page fetches [2]). We verified that there's no correlation between this cost and actual time(i.e., wall clock time) with linear regression which can be seen as an error-minimizing mapping of the optimizer-estimated cost (which is not measured in ms) to CPU-time. The results are not effective and were similar to the ones found in [6, 9]. Therefore it is clear from previous work that post-processing the optimizer cost is not useful. In [16], authors have taken a pre-processing calibration approach wherein they tune system parameters to reflect the underlying hardware.

Recent work [6, 9, 7] has explored the use of machine learning based models for resource/execution time estimation. In these approaches, statistical models are trained on actual execution of training queries with certain set of features [e.g., number of joins, aggregates etc.]. After the pre-processing, these models are usually written to disk to make on-line predictions as and when needed. When given sufficient training data, these statistical models can fit complicated functions and dependencies better than hand-tuned formulas. Because they are trained on actual instances, they can usually capture wide range of effects including special cases in query processing, hardware architecture and database configuration parameters.

1.2 How Robust Are Statistical Models?

While the proposed statistical techniques can improve estimation accuracy significantly they also fail dramatically [7] when the queries are *different* from the ones seen during training. Examples of such differences might be a change in underlying data size, distribution, changes in query plan or a new unseen query altogether. We consider a model to be robust if the differences between training and testing queries do not significantly degrade the estimation accuracy. In the case of static workload where query template seldom changes (e.g., report generation tools where only the query parameters change), this is acceptable. However workloads are often dynamic which implies estimator must be robust to ad-hoc queries.

The issue of dealing with differences between the training and testing data effects any machine learning approach, including the one proposed in this work. So the features need to be extracted in such a way that it should be very unlikely for two different queries to map to similar feature vector. For example consider the approach proposed in [7] where a query features are essentially number of instances of each operator, input and output tuples. It then predicts run time of a new query by averaging the execution times of k nearest queries in feature space. With an extra aggregate in query the run time might increase substantially but changes in feature vector might not be significant [7, 6]. We argue here that the problem is not with

that of learning technique they used but rather with the representation of features. Accurate estimations are difficult at plan level because much of the information about the query plan is collapsed at that level. Hence, there is a need to consider learning at a finer granularity level and construct rich feature sets. By going deeper into query tree, we can ensure that it will be less unlikely for two very different queries to map to similar features.

1.3 Contributions

- We propose statistical techniques which learn query behavior at a finer granularity i.e., at physical implementation level of a query operator. We then compose these individual operator running time estimates to produce an overall execution time estimation for a given query plan.
- Through a benchmark workload we show that the proposed technique outperforms the current state of art.

1.4 Assumptions

Before we go into the details of approach, we list the constraints we imposed 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.
- 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.

The rest of the report is organized as follows: we start with an overview of proposed approach in Section 2. We introduce operator level models and the problems in extracting training data in Section 2.1. We propose a feature that is "pipeline-aware" as a solution in Section 2.2. We then describe the training and testing phases in Sections 2.3 and 2.4, respectively. We validate our approach and present the results in Section 3. We discuss related work and the applicability of their solution to the current problem in Section 4. Next, we discuss the shortcomings of our approach in Section 5. We conclude the report by discussing future work in Section 6.

Chapter 2

Proposed Approach

2.1 Overview

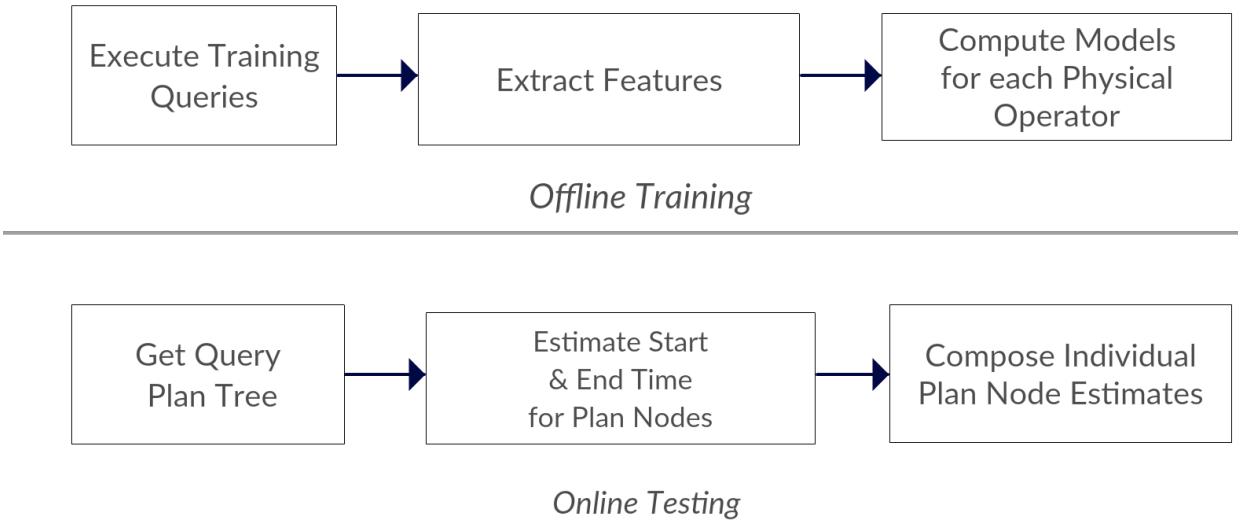

Figure 2.1: Overview of proposed approach

Our approach consists of two distinct phases: an off-line Training and an on-line Testing. During training phase, we execute a specific set of queries on the target system and collect the feature set for each operator. For each physical implementation of an operator, we build two models; start time model to predict the time taken to produce first tuple and end time model to predict the time taken to produce rest of the tuples. We recursively use this information

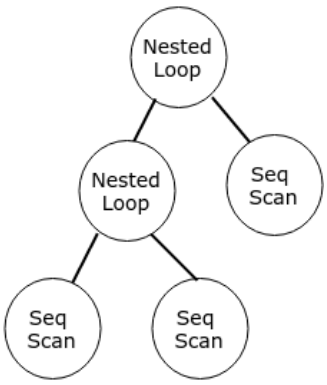

Figure 2.2: Example Query Plan Tree

from child operators to produce an estimate for its parent operator. Hence, the total execution time will be the sum of start and end time of the root node in 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.

2.2 Operator Level Models

It is important that we generalize to arbitrary queries not seen during training. For this purpose, we use the fact that query plan are broken into a set of SQL operators within database engine already. Each SQL operator has multiple physical implementations. Since each of the implementation significantly differ in terms of running time and space consumption, we need to further distinguish them. Therefore we build models for each such implementation of an operator. 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 a operator involves extracting individual running time of an operator from the total execution time. In a non-pipelined environment this is simple and requires no further work. But most databases including PostgeSQL has a pull based execution model which introduces pipelining. Here, operators can be divided into blocking/non-blocking categorized according to their nature of processing input. A blocking operator needs to process all rows before it can pass the data to its parent, while a non-blocking operator passes rows to its parent as soon as they are read and processed. For example, a Nested Loop is a non-blocking operator and Hash is blocking operator since probing cannot begin unless the hash table is entirely constructed. A series of non-blocking operators create a pipeline with output of child operator connected as an input to its parent.

Consider the sample query plan shown in Figure 2.2, both the *Sequential Scan* and *Nested*

Loop operators are non-blocking which allows them to pass on the tuples to their parent as soon as they are read and processed. This creates a pipeline and it speeds up the processing by making use of the possible computation and I/O overlap present in the query. The end-effect of such concurrent behavior on execution time is difficult to capture perfectly but can be approximated to some extent by finding patterns in corpus of training query executions.

To take these subtle differences into consideration, we build two learning models for each operator:

- A Start time prediction model to find the time taken by an operator (including sub-query rooted at this operator) to produce its first tuple. This allows us to capture the blocking nature of operator and help parent operator in deriving its own estimate.
- An End time prediction model to find the time spent in producing the rest of the tuples.

To illustrate the notion of start and end time, consider *Hash-Join* operator with a table scan for outer table. To begin the join operation, we first need to construct the hash table on inner table. After the construction, hash table gets probed for each tuple in outer table. Here the start time of *Hash-Join* operator is the time spent to build the hash table plus the time taken to emit first tuple after a successful probe. End time is difference between the production of last and first tuple. Hence the total time taken by an operator is the sum of start and end times.

2.3 Features

We used a fixed set of features to create models for each operator. The complete list is shown in Table 2.1, values for these features are extracted from PostgreSQL 9.4 *explain analyze* command. These features are applicable to almost all operators. When not applicable (for e.g., leaf nodes which does not have children), values are substituted with a default value.

Feature
Number of left child input tuples
Width of left child input tuple
Number of right child input tuples
Width of right child input tuple
Number of output tuples
Estimated left child start time
Estimated left child end time
Estimated right child start time
Estimated right child end time
Is left child a Blocking operator(0/1)
Is right child a Blocking operator(0/1)

Table 2.1: List of features.

The features shown in Figure 2.1 are based on the domain knowledge of database internals and the features considered by previous work in literature [7, 6]. We have evaluated all the subsets for suitability using Exhaustive subset search w.r.t the metric defined in Section 3.1. The set as a whole produced better accuracy compared to any of the proper subset.

However this list is no way exhaustive as query plans usually contain many more attributes often very specific to an operator. For example, seq scan with a filter takes different amount of time based on the evaluation complexity of predicate and number of those predicates. More features usually attribute to better accuracies, and hence using operator specific features can further improve accuracy. However the amount of training data needed is directly proportional to the number of features. So there's a implicit trade-off here. In this work, we limit ourselves to a fixed number of features.

Having described the intuition behind 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.

2.4 Training

In this phase, prediction models are derived by executing set of queries and observing their true running times. More precisely, example instances of both start and run 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, 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. For example, in case of a table scan optimizer chooses Index scan when fewer no. of tuples qualify and a seq scan when there are beyond a certain no. of tuples.
- Each operator has sufficient number of example instances. For the current feature set with 10 features, we need to have approximately a thousand 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. We have tried multiple predictions models including linear, non-linear kernels in Support Vector Regression (SVR) and Tree based models. The linear models fared worst among all, because of their inability to model non-linear relations among features. Tree based models such as Gradient Boosting performed the best, however their learning times are beyond reasonable even with that of a scalable and parallel implementation (XGBoost, available at [5]). Tree based models also lack the ability to "extrapolate" which limits the generalizing ability of model. Therefore we have settled with SVR using Radial Basis Function (RBF) as its kernel. We demonstrate the accuracy gains possible with non-linear kernel over linear kernel by showing prediction results for each operator based on internal cross validation. For complete information, refer the Appendix Tables 1, 2. Note that SVR need to be tuned properly to fit the underlying data. 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. The behavior of the model is very sensitive to the gamma parameter. 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. We used Grid search to train SVR for each pair (C, γ) in the cartesian product of these two sets and evaluated their performance by internal cross-validation on the training set. At the end, grid search algorithm outputs the settings that achieved the highest score (w.r.t metric defined in Section 3.1) in the validation procedure. Model with the best hyper-parameter values will be trained on entire training set. Grid search is embarrassingly parallel because the hyper-parameter settings it evaluates are independent of each other. So, we have parallelized the grid search in Python and were able to obtain speedup proportional to the number of cores. In this way, for every operator we learn two prediction models and materialize them to disk. 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 overall space consumption (including models and scaling functions) is around 4MB.

With the pace hardware speeds and predictions model are evolving, it is very well possible that a new prediction model can have the best of prediction accuracies as well as the running times (e.g., Recurrent neural networks [4]). It would be interesting to see impact of those techniques when embedded in this framework.

2.5 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(s) to get an estimate. These estimates are in-turn used by parents to produce their own estimate. The estimated execution time of a given query plan is therefore the sum of start and end prediction times of *root* node.

Special cases like leaf and unary nodes are handled by assuming a default value for the unavailable features. For example, in case of a unary operator like Sort we assume a zero value for number of tuples in right child. Note that this does not impact the prediction accuracy as features with zero variance are not considered during evaluation.

Chapter 3

Experimental Evaluation

In this section we evaluate the accuracy and robustness of our technique along with the current state of the art [16]. To evaluate the robustness we have specifically taken test queries which are different from that of training.

3.1 Setup

- Database management system: PostgreSQL 9.4 [1]. Please note that the earlier Postgres version lack the instrumental features that training requires (such as actual running time of node).

- Datasets and Workload: We have used TPC-H benchmark [3] queries for training the predictive models. We generated underlying data distribution by a tool published at [11]; 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. We have used QGEN [3] tool to generate 20 instances for each query template; producing a total of 420 queries for a total of 21 query templates. For enabling prediction models to learn the effect of the scale of data we have ran these queries across 0.1, 0.5, 1, 2 and 3 GB. Therefore, our training set had a total of 2100 queries. For testing, we have separately generated one query for each TPC-H template and used a larger 5 and 10GB workload.

- 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 and database buffers.

- Predictive models: We have used Python's sckit-learn [12] library for all the ML implementations.
- Error Metric: We have used the Q-Error (QE) [8, 10] 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 [16, 6, 9], 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.

- Alternative techniques: We compare the accuracy and generalization ability of tuning approach proposed in [16]. As in their case, we will assume perfect selectivity estimates while producing the results. For ease of reference, we refer their [16] approach as Tuning and our approach as Learning.

3.2 Evaluation

In this experiment we have generated the test queries using QGEN tool. We made sure that the training and testing queries do not contain identical query (i.e., same template and instance). We have shown the per query error in Table 3.1 and 3.3 for 5 and 10GB respectively.

In Table 3.2 and 3.4, we show the summary statistics. For 5GB data-set, Learning outperformed Tuning approach by "1.9x". Intuitively, AQE of 2.1 indicates that on an average the predictions were within twice/half of actual execution time. In the case of 10GB data-set, both the learning and tuning performed similarly and there were no accuracy gains using Learning here. In our approach, we observed that errors made by individual operators are propagating and getting multiplied because of the way features are modelled recursively. This problem does not impact the Tuning approach in the same magnitude because errors made by operators are independent with respect to each other.

Query	Actual(ms)	Tuning(ms)	Learning(ms)	QE(Tuning)	QE(Learning)
1	118801	87412	94990	1.35	1.25
3	24680	65927	46382	2.67	1.87
4	614	1138	901	1.85	1.46
5	80482	72251	47730	1.11	1.68
6	46634	6891	57117	6.76	1.22
7	61630	66987	62918	1.08	1.02
8	142814	71306	51323	2.00	2.78
9	68407	61606	74627	1.11	1.09
10	53056	54876	8063	1.034	6.57
11	6415	107594	1211	16.77	5.29
13	20802	77402	17466	3.720	1.191
14	2132	30934	4364	14.50	2.04
15	7008	26640	7732	3.80	1.1
19	37275	9516	34764	3.91	1.072
20	44664	74347	65019	1.66	1.45
21	13315	3994	3622	3.33	3.67

Table 3.1: Per Query running time w.r.t TPC-H (5GB)

	Tuning	Learning
Average	3.98	2.10
Minimum	1	1
Maximum	16.77	6.5

Table 3.2: Learning and Tuning comparison w.r.t QE, TPC-H (5GB)

Query	Actual(ms)	Tuning(ms)	Learning(ms)	QE(Tuning)	QE(Learning)
1	247585	187922	88871	1.31	2.78
4	195173	37761	56968	5.16	3.42
5	991958	51398	65201	19.29	15.21
6	88512	110234	5093	1.24	17.37
7	182561	133116	66095	1.37	2.76
8	403421	79910	71331	5.04	5.65
9	1414600	155381	57690	9.1	24.52
10	205293	133743	128747	1.53	1.59
11	230688	3968	12299	58.12	18.75
13	743406	35508	63888	20.93	11.63
14	98107	95596	32181	1.02	3.04
15	101742	22418	115193	4.53	1.13
16	399465	639420	31805	1.6	12.55
17	220425	518937	31965	2.35	6.89
18	87808	136058	31784	1.54	2.76
21	126832	8216	7956	15.43	15.94

Table 3.3: Per query running time w.r.t TPC-H (10GB)

	Tuning	Learning
Average	9.12	9.35
Minimum	1.24	1.13
Maximum	58.12	24.52

Table 3.4: Learning and Tuning comparison w.r.t QE, TPC-H (10GB)

Chapter 4

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 [6, 9], as well as in the context of interactions between concurrently executing queries [15].
The work done in [7] is the first of its kind to embed Machine learning techniques inside query optimizer. However it has lot of limitations on its applicability:

- The 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 after mapping Qs feature vector using Kernel Canonical Correlation Analysis (KCCA) into a suitable similarity space. This becomes a problem when we are trying to estimate the resource consumption of a significantly expensive query than all the training queries, the estimated value can never be larger than the ones encountered during queries. Thus, this technique is not capable of "extrapolating" beyond the training data.

- It models the queries at Plan level with set of key, value pairs as features. Every physical database operator corresponds to a key and the sum of all the input cardinalities for that operator in the query plan corresponds to the value. This makes the approach vulnerable to changes between training and test data not encoded in this feature set e.g., when the training and test queries use different schemas and databases, the estimated run-times were up to multiple orders of magnitude longer than the actual time the queries ran (as shown in Figure 15 in [7]). 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 [6] mitigates this issue to some extent by introducing operator 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. The authors use linear regression models for each operator; that means they implicitly force the output to vary linearly with each input feature. In reality, the relationship between features and execution time is non-linear.

The work of [9] explores the use of powerful statistical learning techniques such as 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). Their approach is limited to database with a "push based execution model" where child nodes execute completely before passing their data to parent nodes. Therefore, they cannot straightaway work for database systems such as PostgreSQL without accounting for its style of execution.

The approach proposed in [16] 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 best accuracy making it the current state-of-art. However being analytical they lack the power to account for the query optimization within operators and 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.

Finally our work is also related to [8] in the context of studying the effect of cardinality estimators and cost model on the overall execution time estimation. Through a set of join order benchmark queries (JOB) they show that cardinality estimates often introduce magnitude of errors that they outweigh the errors introduced by cost model.

4.1 Limitations

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

- Pre-processing overheads: We have a one-time training and computing phase which takes considerable amount of time to finish.

- Implementation environment: We currently implement the estimation framework outside the PostgreSQL database engine in Python. For it to be applicable to query optimizer, we need to implement this inside database engine in C programming language. However currently C language has very limited support for complex ML libraries.

Chapter 5

Conclusion

In this work, we studied the effect of cost model on overall execution time estimation. We have shown that with true selectivity estimates in place we can achieve either similar or better predictions. We achieve these improvements by using a pipeline aware feature set along with a powerful learning model that can account for the complex dependencies. By modeling at the level of physical implementation of an operator, we have shown that predictions for ad-hoc queries can be achieved.

Chapter 6

Future Work

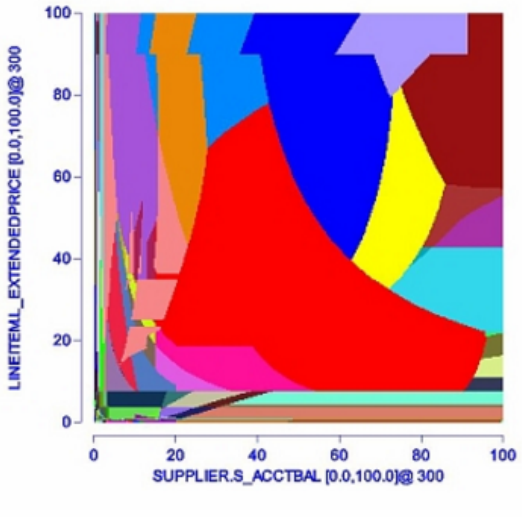

Figure 6.1: Example Plan Diagram

We believe this work opens up lot of opportunities to further improvise the estimates. For e.g., in training phase we wish to replace ad-hoc learning by Incremental Learning; currently we rely on TPC-H skew generation tool [11] to produce query instances that hope to have a varying resource consumption among the same query template. Instead, we can *manually* explore through the error prone selectivity space. For example, consider the plan diagram shown Figure 6.1 produced by the Picasso tool [13], where each color represents a different plan. It can be seen that just for a single query there are as many as 109 query plans. To extract these query plans, we need to systematically explore the 2-dimensional selectivity space i.e., $[0, 100] \times [0, 100]$ at a fixed step-size/resolution. At every such point in the space we find the corresponding query

instance and add it to training set. This way, we can efficiently "discover" different query plans available in plan space which reduces the number of training queries required (consequently pre-processing time) and can possibly produce better models. With such learning system in place, it would be interesting to study the relation between accuracy and number of training examples which can help us find the saturation point.

Appendices

Evaluation of Linear Kernel

Operator	Min QE	Max QE	Avg QE
sSort-top-Nheapsort	1.00	1.01	1.01
sMaterialize	7.85	55.64	31.16
eBitmapOr	1.03	1.31	1.15
eNestedLoopNestedLoop	1.00	1.02	1.01
sHashJoin	3.01	18.86	7.45
eNestedLoopBitmapHeapScan	1.25	1.55	1.39
sAggregate-Hashed	1.57	35.70	8.48
eNestedLoopIndexOnlyScan	1.35	55.97	17.84
eMaterialize	2.15	41.58	15.58
sNestedLoopIndexScan	2.06	17.48	5.45
eHash	1.01	1.03	1.02
sBitmapAnd	1.08	49.68	16.20
sAggregate-Sorted	19.25	79.28	36.13
sLimit	1.01	3.32	1.48
sNestedLoopIndexOnlyScan	1.02	1.08	1.05
eSeqScan	1.77	2.51	2.01
eSort-top-Nheapsort	1.00	1.01	1.01
eNestedLoopMaterialize	2.71	5.70	3.67
eSort-quicksort	1.03	14.38	4.05
eNestedLoopIndexScan	34.55	76.34	54.89
eAggregate-Plain	1.08	1.17	1.12
sAggregate-Plain	1.08	1.17	1.12
eSort-externalmerge	1.23	4.04	2.60
eAggregate-Sorted	1.00	1.01	1.01
eSort-externalsort	1.20	3.61	2.09

sNestedLoopNestedLoop	1.00	1.12	1.05
eLimit	1.01	3.32	1.48
sHash	1.01	1.03	1.02
sNestedLoopMaterialize	2.94	34.84	12.37
sSort-quicksort	1.01	7.96	3.49
sMergeJoin	109.91	890.37	525.76
eNestedLoopSeqScan	1.00	2.28	1.27
eBitmapHeapScan	10.26	18.60	16.10
sBitmapIndexScan	2.67	12.14	7.44
sNestedLoopBitmapHeapScan	1.35	3.37	2.01
sSeqScan	15.57	529.31	190.85
eAggregate-Hashed	1.57	36.43	8.62
eHashJoin	2.06	22.55	7.42
eIndexScan	6.13	9.26	7.63
sNestedLoopSeqScan	1.00	7.61	2.77
sBitmapHeapScan	1.23	1.80	1.51
sIndexOnlyScan	1.46	3.58	2.51
eBitmapIndexScan	2.67	12.14	7.44
eMergeJoin	4.75	56.15	17.20
sSort-externalmerge	1.23	3.54	2.41
sIndexScan	2.31	10.36	4.94
sBitmapOr	1.03	1.31	1.15
eIndexOnlyScan	1.07	10.22	3.05
sSort-externalsort	1.21	3.67	2.13
eBitmapAnd	1.08	49.68	16.20

Table 1: QE for each operator using Linear kernel

Evaluation of Non-Linear Kernel

Operator	Min QE	Max QE	Avg QE
sSort-top-Nheapsort	1.01	1.10	1.04
sMaterialize	4.11	90.26	43.66
eBitmapOr	1.03	1.31	1.15
eNestedLoopNestedLoop	1.00	1.24	1.10
sHashJoin	7.34	31.33	20.07
eNestedLoopBitmapHeapScan	1.09	1.92	1.38
sAggregate-Hashed	1.05	20.16	4.96
eNestedLoopIndexOnlyScan	1.36	58.20	18.45
eMaterialize	2.19	20.94	10.33
sNestedLoopIndexScan	12.22	30.14	19.32
eHash	1.11	1.69	1.36
sBitmapAnd	1.06	47.65	10.97
sAggregate-Sorted	15.22	37.21	24.69
sLimit	1.00	1.79	1.18
sNestedLoopIndexOnlyScan	1.04	2.59	1.42
eSeqScan	2.01	2.76	2.35
eSort-top-Nheapsort	1.01	1.10	1.04
eNestedLoopMaterialize	1.42	50.28	14.56
eSort-quicksort	1.05	4.05	1.72
eNestedLoopIndexScan	3.04	333.78	86.75
eAggregate-Plain	1.04	1.56	1.18
sAggregate-Plain	1.04	1.50	1.15
eSort-externalmerge	1.66	3.37	2.67
eAggregate-Sorted	1.00	1.04	1.02
eSort-externalsort	3.12	17.51	8.49

sNestedLoopNestedLoop	1.00	1.26	1.11
eLimit	1.00	1.79	1.18
sHash	1.11	1.69	1.36
sNestedLoopMaterialize	7.24	20.36	13.96
sSort-quicksort	1.05	4.08	1.73
sMergeJoin	17.64	254.71	134.49
eNestedLoopSeqScan	1.00	17.82	5.81
eBitmapHeapScan	3.20	92.47	22.18
sBitmapIndexScan	4.52	14.12	9.21
sNestedLoopBitmapHeapScan	1.21	2.75	1.73
sSeqScan	23.08	616.10	237.35
eAggregate-Hashed	1.05	19.05	4.73
eHashJoin	1.07	11.29	3.36
eIndexScan	5.82	10.71	7.95
sNestedLoopSeqScan	1.00	14.30	4.88
sBitmapHeapScan	1.37	17.21	4.77
sIndexOnlyScan	1.49	9.53	4.22
eBitmapIndexScan	4.52	14.12	9.21
eMergeJoin	2.24	24.58	7.44
sSort-externalmerge	1.58	3.13	2.48
sIndexScan	2.28	10.33	5.20
sBitmapOr	1.03	1.31	1.15
eIndexOnlyScan	1.61	9.06	3.17
sSort-externalsort	3.17	18.49	8.91
eBitmapAnd	1.06	47.65	10.97

Table 2: QE for each operator using RBF kernel

References

[1] Postgresql 9.4 dbms. URL https://www.postgresql.org/docs/9.4/static/release-9-4.html. ii, 12

[2] Postgresql cost model. URL https://www.postgresql.org/docs/9.4/static/sql-explain.html. 3

[3] Tpc-h benchmark specification. URL http://www.tpc.org/tpch/. ii, 12

[4] Recurrent neural network. URL https://en.wikipedia.org/wiki/Recurrent_neural_network. 10

[5] Scalable and flexible gradient boosting. URL https://xgboost.readthedocs.io/en/latest/. 9

[6] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. Learning-based query performance modeling and prediction. 2012. 3, 8, 13, 16

[7] 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 *ICDE*, 2009. 3, 8, 16

[8] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *VLDB*, 2015. 2, 13, 17

[9] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *VLDB*, 2012. 3, 13, 16, 17

[10] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *VLDB*, 2009. 13

[11] Vivek Narasayya. Program for tpc-h data generation with skew. URL http://research.microsoft.com/en-us/downloads/98710c69-8db6-4a59-a217-6651c8aabbf4/. 12, 19

[12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 2011. 13

[13] Naveen Reddy and Jayant Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, 2005. 19

[14] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, 2001. ii, 2

[15] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *VLDB*, 2013. 16

[16] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? 2013. 3, 12, 13, 17