Prediction of Index Scan Execution Time

Vishal Goel

Mid-term MTech Project Report

Abstract

Prediction of query execution time has been a problem of vital importance since the very time the database management systems came into existence. Knowing how long a query is expected to execute not only helps in query optimisation but also in taking decisions concerning admission control and query scheduling. Considering how important and widely known the problem is, surprisingly there has not been much work in this space in the past. The problem is challenging particularly because it involves predicting the runtime environment like anticipating data access pattern, spilling, etc. The literature involves two primary approaches – (a) machine learning based models and (b) parameter tuning approaches to existing models. While the former is expensive and lacks explainability, the latter achieves a limited accuracy. In this work we attempt to design a new cost model for PostgreSQL engine while addressing the above issues. In particular, we take a look into the cost models of scan operators and show that even though a well-tuned PostgreSQL engine predicts sequential scan correctly, it does not predict index scan that well. Further, we propose a preliminary polynomial regression model for index scan to handle queries involving range predicates. Our experiments show that the proposed model produces better results than well-tuned PostgreSQL engine.

1 INTRODUCTION

Prediction of query execution time is a classical problem in the query processing literature. Other than the obvious benefit of knowing when the query would finish executing, it has several other use cases:

Query Optimisation: To pick the least cost (estimated execution time) plan to execute a given query, query optimiser estimates the output row cardinality of each operator in the plan using the cardinality estimation model and its associated cost using the cost model.

Query Scheduling and Admission Control: Knowing execution time of a query helps in latency-aware scheduling and making deadline-based decisions.

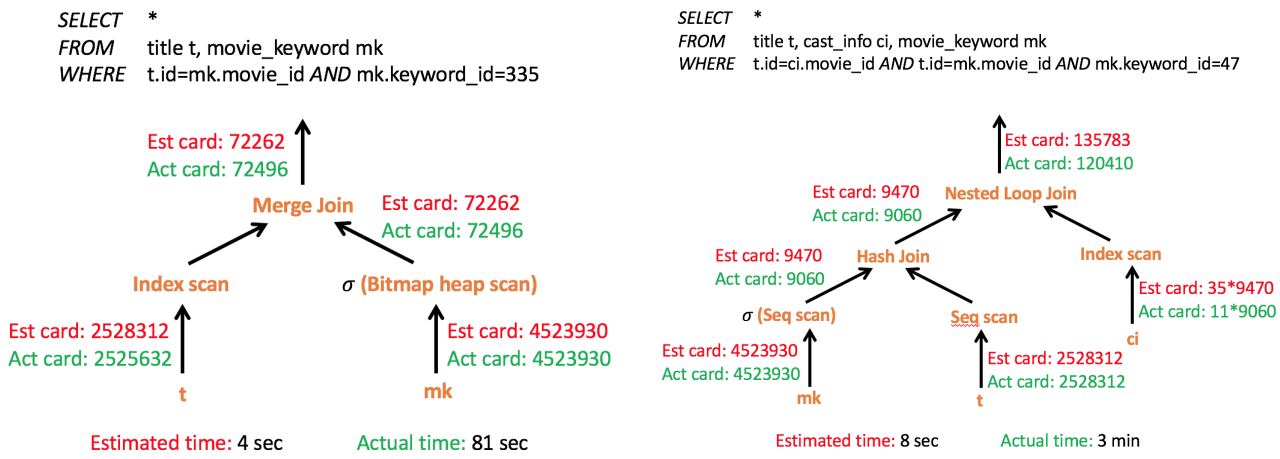
Progress Monitoring: Keeping a track of execution progress of queries.

System Sizing: Prediction of query execution time as a function of hardware resources helps in picking the right environment for query execution.

The recent work in this space (in the last decade) has followed two primary approaches: machine learning (ML) ([1], [2], [3], [5], [6]) and statistical tuning ([8], [9]). The ML-based approaches, explained in detail in Section 7, look at the database engine as a black box and try to predict execution time of a given query using ML models like linear regressors, SVMs, decision trees and even neural networks. The statistical tuning technique[1] by Wu et al[8], however, simply tunes the cost parameters of PostgreSQL's cost model by running a set of calibrating queries. While ML models are expensive and lack explainability, statistical tuning provides limited improvement in predictions, as discussed in Section 2.

To address the above issues, we are working on building a new cost model. In this report, we talk about the scan operators (one of the most critical operators, as discussed in Section 3) and analyse their costing approach in PostgreSQL. We found that while PostgreSQL correctly estimates sequential scan operator, the costing of index scan operator has significant shortcomings. Further, we present a preliminary ap-

[1]Statistical tuning is also an ML technique but it is different in the regard that it focuses only on tuning the cost parameters of PostgreSQL's native cost model (assuming the model is reasonable) instead of building afresh.



(a) Prediction by PostgreSQL without tuned parameters

(b) Prediction by PostgreSQL with tuned parameters

Figure 1: Example Query Plans

proach towards building a better cost model and show experimentally that even using a simple polynomial regression approach can improve results significantly.

In the report below, Section 2 provides an overview of PostgreSQL's cost model, the statistical tuning approach to tune its cost parameters. Section 3 is a comment on the importance of predicting scan operators and an introduction to sequential and index scan operators. Section 4 presents the cost model of index scan in PostgreSQL and its limitations. Further, Section 5 presents the polynomial regression approach to predict index scan execution time followed by experiments in Section 6, a brief explanation of related work in Section 7, and conclusions and future work in Section 8.

2 Cost Model of PostgreSQL

Postgres cost model uses a vector of five cost parameters: $c = [cs, cr, ct, ci, co]^T$, defined as follows:

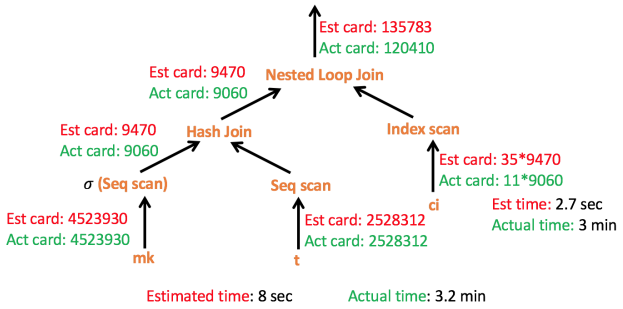
1. cs: I/O cost to sequentially access a page
2. cr: I/O cost to randomly access a page
3. ct: CPU cost to process a relation tuple
4. ci: CPU cost to process an index tuple
5. co: CPU cost to perform an operation like hash and aggregation

The cost C_O of an operator O in a query plan is given by $C_O = n^T c$ where $n = [ns, nr, nt, ni, no]^T$ represents the number of pages sequentially accessed, number of pages randomly accessed, number of relation tuples processed, number of index tuples processed and number of CPU operations respectively, all during the execution of the operator O.

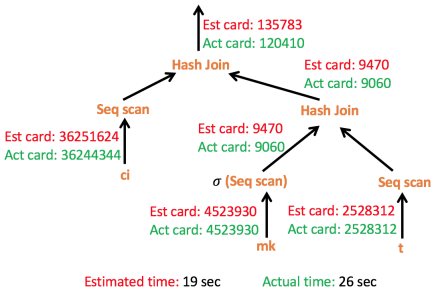
Statistical Tuning: Wu et al [8] tune these cost parameters using a set of calibrating queries over a set of tables. Figure 1a is an example of a query plan where PostgreSQL estimated the execution time of 81 secs (actual time:4 secs) but the tuning fixed the estimation. However, the Wu-tuned PostgreSQL did not do good for another query shown in Figure 1b (estimated: 8 secs, actual: 3 mins). Note that bad cost estimates can have a significant impact on query optimisation, as shown in Figure 2. When the query Q (in Figure 2a was fired on the well-tuned engine, plan A (Figure 2b) was picked over plan B (Figure 2c). This is because the optimiser underestimated the cost of index scan (2.7 secs instead of 3 mins). Note that in plan A, the input cardinality to the index scan operator was overestimated and, thus, with perfect input cardinality, the estimation will be even lesser. Further, this experiment is a proof that it is not possible to multiply the estimated cost given by PostgreSQL with a magic number to get the actual time because the estimated time of plan A is less than estimated time of plan B but actual time of plan A is greater than actual time of plan B.

```
SELECT *
FROM title t, cast_info ci, movie_keyword mk
WHERE t.id=ci.movie_id AND t.id=mk.movie_id AND mk.keyword_id=47
```

(a) SQL Query Q



(b) Plan A (picked by Wu-tuned Postgres for query Q)



(c) Plan B (an alternative better plan for query Q)

Figure 2: An example of how bad prediction can affect query optimisation

3 Prediction of Scan Operators

We focus only on predicting the scan operators for hard-disk-resident databases. Scan operators typically take up most of the total query execution time (more so in the case of disk-resident databases) and, thus, their prediction is imperative.

There are two types of scan techniques that DBMSs adopt – sequential scan and index scan. On running around 20 random queries from Join Order Benchmark (JOB) [4], we found that 96% of the total query execution time was spent in scan. Further, 88% of the total scan time was spent in index scan.

```
SELECT *
FROM R
WHERE v1 ≤ A ≤ v2
```

Figure 3: Query Template $Q\langle R, A, v1, v2\rangle$

We will now discuss both the scan operators in detail. For the sake of convenience, let us assume a SQL query $Q\langle R, A, v1, v2\rangle$ as shown in Figure 3. Executing Q returns the tuples from relation R whose values of attribute A lie between v1 and v2 (both inclusive).

3.1 Sequential Scan Operator

Given a query $Q\langle R, A, v1, v2\rangle$, PostgreSQL reads the first page (a random access because the disk head can be anywhere initially) and then sequentially accesses the subsequent pages of the relation.

$$Estimated\ Cost = cr + (n-1) * cs$$

where n = total pages in the relation.

We evaluated Wu-tuned PostgreSQL optimiser's predictions on all tables in JOB for sequential scan and all the queries were found to have relative error of less than 1%.

3.2 Index Scan Operator

There can be two types of predicates in a query: equality predicates ($A = v$) and range predicates ($v1 \leq A \leq v2$). Currently, we have focused only on range predicates.

During index scan, PostgreSQL first reads the pages of the index from root to the first leaf (containing value $v1$), then reads the data pages corresponding to the values present in that leaf (in order), then reads the next leaf page and repeats the procedure until the relation page with the last value $v2$ has been read. Note that pages once accessed from disk may be found cached next time. Thus, number of disk accesses for data pages are typically not equal to the number of tuples in the range $[v1, v2]$, unless the buffer size is too small. The index stores values of the attribute in ascending order at the leaf level and thus, the data pages can be found at random locations on the disk, unless the index is clustered (i.e. the order of attribute values in index is same as order of attribute values in the relation), in which case index scan simply acts as several partial sequential scans on the table.

Range (in 10^5)	Est Relation Pages	Est Index Pages	Est Total Cost	Actual Relation Pages	Actual Index Pages	Est Index IO cost	Est Relation IO cost	Actual Time	Est Total Cost / Actual Time
<1	9367	3684	7 secs	9694	3760	6.5 secs	0.2 secs	0.6 secs	0.085
<10	94800	37279	70.7 secs	96206	37178	66.3 secs	2.6 secs	6 secs	0.084
10 to 20	99663	39192	74.7 secs	100448	39271	69.7 secs	2.8 secs	6.3 secs	0.084
>20	58226	22913	43.5 secs	56076	22594	40.8 secs	1.6 secs	3.5 secs	0.080
Full range	252654	99382	188 secs	252654	99382	176.9 secs	7 secs	15.9 secs	0.084

(a) $CF = 1 : select * from\ cast\_info\_sorted\ where\ v1 \leq movie\_id \leq v2$

Range (in 10^5)	Est Relation Pages	Est Index Pages	Est Total Cost	Actual Relation Pages	Actual Index Pages	Est Index IO cost	Est Relation IO cost	Actual Time	Est Total Cost / Actual Time
<10	95148	37416	71 secs	96207	37178	66.8 secs	2.6 secs	69.5 secs	1.02
10 to 20	100064	39349	75 secs	100446	39267	70 secs	2.8 secs	210 secs	0.35
>20	57335	22547	43 secs	56074	22584	40 secs	1.6 secs	294.8 secs	0.14
Full range	252654	99382	188 secs	252654	99382	177 secs	7 secs	570 secs	0.33

(b) $CF = -1 : select * from\ cast\_info\_sorted\_desc\ where\ v1 \leq movie\_id \leq v2$

Range (in 10^4)	Est relation pages	Est index pages	Est total cost	Actual relation pages	Actual index pages	Actual CF	Est cost with actual relation pages and actual index pages	Est cost with actual relation pages, actual index pages and actual correlation	Actual time	Est Total Cost / Actual Time
44 to 45	117567	421	210.8 secs	29511	395	-0.34	~ 51 sec	~ 45 sec	105.6 secs	1.99
8 to 9	101431	348	181.8 secs	17743	377	-0.26	~ 31 sec	~ 28 sec	63.3 secs	2.87
<1	118771	426	213 secs	32889	430	-0.14	~ 57 sec	~ 55.8 sec	120.5 secs	1.76
68 to 69	94219	318	169 secs	21318	269	0.001	~ 38 sec	~ 38 sec	68 secs	2.48
1 to 4	211146	995	379 secs	63305	988	0.03	~ 2 min	~ 1.6 min	232 secs	1.63

(c) $CF = 0 : select * from\ cast\_info\ where\ v1 \leq movie\_id \leq v2$

Figure 4: Experimental analysis of PostgreSQL's cost model for index scan

4 PostgreSQL's Cost Model of Index Scan

To keep it brief, we have omitted the parts of the model that estimate the time spent in reading index pages from root to leaf, and the CPU cost of processing the index and relation tuples, both of which are negligible compared to the I/O time of the relation pages (interspersed with I/O time of index leaf pages).

For each index on each relation, PostgreSQL maintains a correlation factor $CF \in [-1, 1]$ between the attribute values and their respective data page ids, which is essentially a measure of the randomness in the data layout of that particular attribute. $CF = 1$ means the index is clustered (order of attribute values in index being same as order of attribute values in the relation i.e. in ascending order). $CF = -1$ means descending order and $CF = 0$ means maximum randomness in the data layout of the attribute. Using CF value, PostgreSQL interpolates between the minimum $MinC$ (cost of scanning pages sequentially) and maximum cost $MaxC$ (cost of accessing pages from random locations) possible to estimate the cost $C(R)$ of index scan on relation R using the following formula:

$$C(R) = MaxC + CF^2 * (MinC - MaxC)$$

where $MinC$ and $MaxC$ are estimated as follows:

$$MinC = relation\_selectivity * total\_relation\_pages * cs$$

$$MaxC = estimated\_relation\_pages * cr$$

and $estimated\_relation\_pages$ is found by the Mackert and Lohman (M&L) approximation model [7]. Further, estimated cost to read index pages expressed as $C(I)$ is given as:

$$C(I) = relation\_selectivity * total\_index\_pages * cr$$

Thus, total cost (TC) to read index and relation pages is given as:

$$TC = C(R) + C(I)$$

Shortcomings in cost modeling

The above cost model for index scan suffers from following critical limitations:

1. It treats all ranges equally by considering global parameters like CF and cr which could be quite different in the local range of the given query.
2. It does not distinguish between the positive and negative CF values, whereas the expected behaviour for both cases is different due to rotation of hard disk in a single direction.
3. The M&L formula (to take caching into effect) also does not work well, as shown in experiments below.
4. The cost model for index scan itself is questionable as it gives inaccurate estimations even after fixing all the above input parameters correctly.

To highlight the above shortcomings, we conducted a few experiments on tables with different data layouts.

Experimental Setup: We selected $cast\_info$ table (6.4 GB) from the IMDB dataset and built an index (776 MB) on the attribute $movie\_id$ ($CF \approx 0$). We also created a copy of the relation called $cast\_info\_sorted$ with $movie\_id$ sorted in ascending order ($CF = 1$) and $cast\_info\_sorted\_desc$ with $movie\_id$ sorted in descending order ($CF = -1$).

To take care of perfect cardinality estimate inputs, the metadata statistics parameters were tuned appropriately so that any errors incurred in estimated cost due to cardinality estimation made a difference of at most one second in all the experiments.

Errors due to global parameters: Figure 4a shows execution statistics of a few queries for $CF = 1$ case. As we can see from the table, PostgreSQL estimated the relation pages and index pages close to accurate and yet, the estimated times are quite far from the actual execution times. Further, since it is a clustered index case, the global and local CF values are the same. As the index scan here effectively converts to sequential scan (interspersed with index leaf page accesses), the relation scan cost ($MinC$ in this case) was estimated fine but the index scan cost was heavily over-estimated. Since number of index pages were estimated correctly, the problem was in using the global cr value here. Note that the estimated total cost vs actual time ratio is nearly same for all queries and thus, a global but right value of cr can still get estimations right here, as we would expect. However, a cr value favourable to one table/index may not be suitable for another.

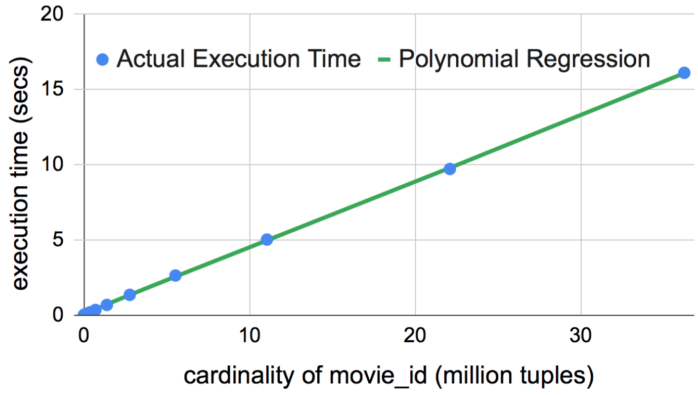
Errors due to neglecting sign of CF: Figure 4a and Figure 4b show performance of PostgreSQL for $CF = 1$ and $CF = -1$ cases respectively. From the last four rows of the tables we can see that the estimated query execution time is identical while the actual times vary drastically. This is expected since the cost function does not take sign of CF into consideration.

Errors due to M&L formula: As we can see from Figure 4c i.e. $CF = 0$ case, PostgreSQL overestimated the relation pages using the M&L formula, making the use of M&L formula questionable.

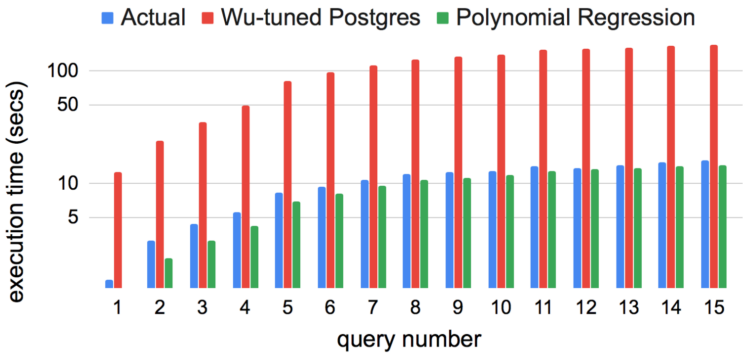
Erroneous Model Design: We can see from Figure 4c ($CF = 0$ case) that even after replacing incorrectly estimated values of the model parameters, i.e. the number of relation pages, index pages and CF with the correct values, the estimated times using the cost model are far from the actual times. This suggests that other than the errors in the estimation of the above three parameters, the model design itself is erroneous. Further, if we take the ratio of estimated time (with the correct input parameters) to the actual time, we do not get a constant implying that a simple constant multiplication factor with the estimated time can also not fix these errors.

5 Polynomial Regression

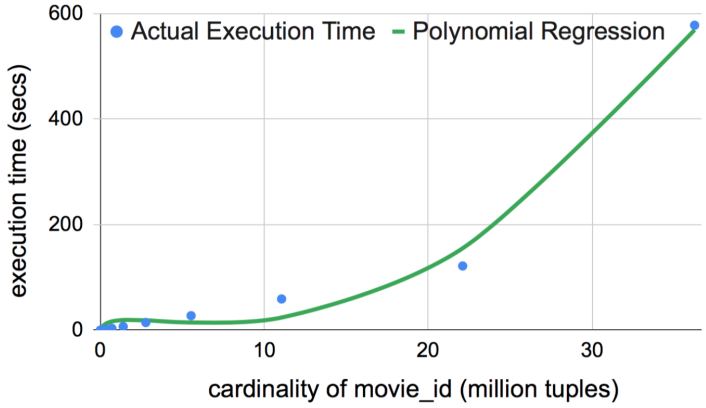
To build a better cost model, it is important to make no assumptions on the data layout. It is expected that if the execution times of all $Q\langle R, A, c, v2\rangle$-type queries ($c$ is a constant) were plotted on a graph (with cardinality of $[c, v2]$ on x-axis and execution times on y-axis), the curve would be monotonically increasing and look like a series of line segments with different (positive) slopes connected end-to-end, irrespective of the data layout. Furthermore, as the number of pages being cached increase towards the end, most of the new requests would be for already cached pages. Hence, the curve is speculated to have an effectively decreasing slope.



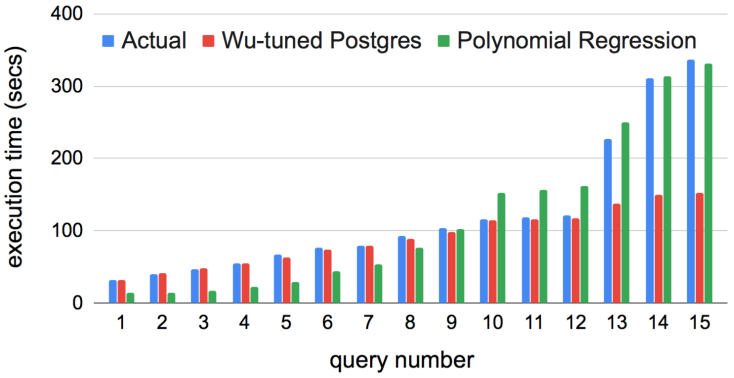
(a) Trained model for $CF = 1$



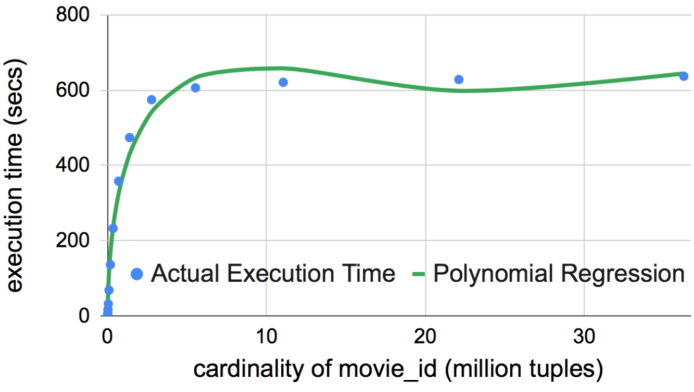
(b) Prediction on test queries for $CF = 1$



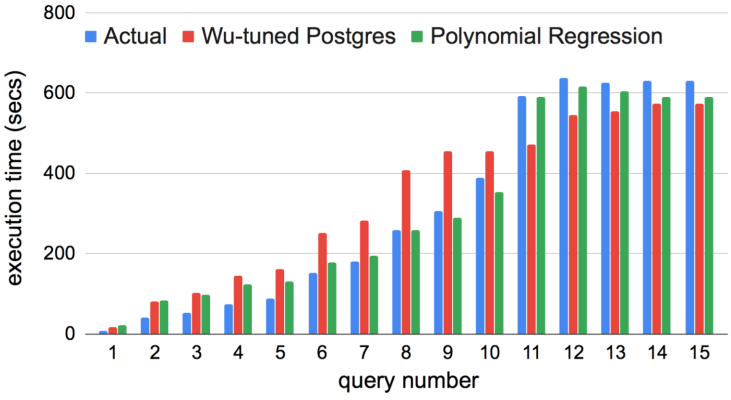
(c) Trained model for $CF = -1$



(d) Prediction on test queries for $CF = -1$



(e) Trained model for $CF = 0$



(f) Prediction on test queries for $CF = 0$

Figure 5: Trained models on different CF values and their predictions on $Q\langle R, movie_id, 1, v2\rangle$-type test queries

Thus, for starters, we have decided to fit a polynomial regressor for $Q\langle R, A, c, v2\rangle$-type queries.

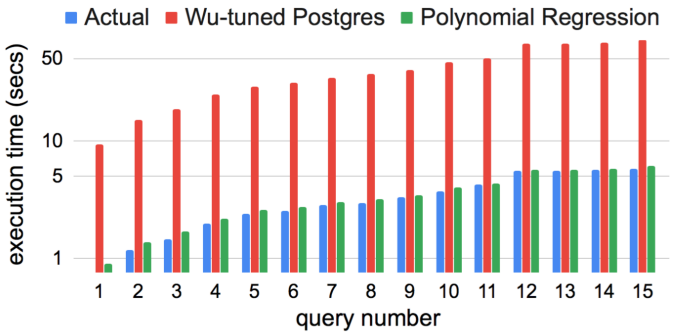
6 Experiments

We have run the experiments for $c = 1$ case for $Q\langle R, movie_id, c, v2\rangle$-type queries where $R = cast\_info/cast\_info\_sorted/cast\_info\_sorted\_desc$. The memory buffers were kept large enough to avoid page replacement. The polynomial regressor was trained on a training data of $log_2 n$ queries (n = total tuples in the relation) for all the three tables. The $log_2 n$ training queries are over a geometric sequence of cardinality values with common ratio 2. This way, out of possible 3.5 million (total tuples in $cast\_info$) training queries, only 22 training queries are required. Thus, the training time was very less (a few minutes in total for three indexes), space required to keep the models is O(1) and the prediction time is O(1). The polynomial regressor was chosen to be of the form

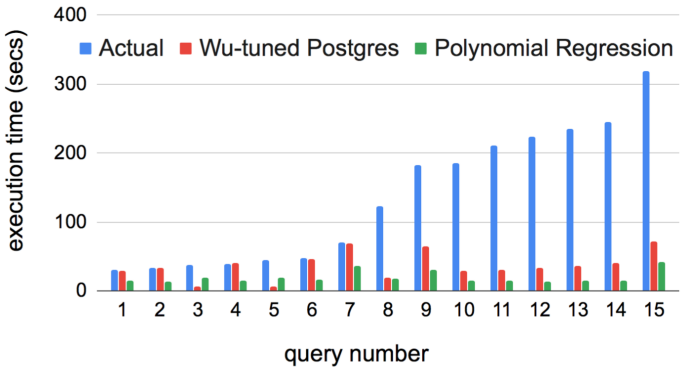
$$y_{prediction} = w_1 * x^{0.5} + w_2 * x + w_3 * x^2$$

where x = cardinality in range $[1, v2]$.

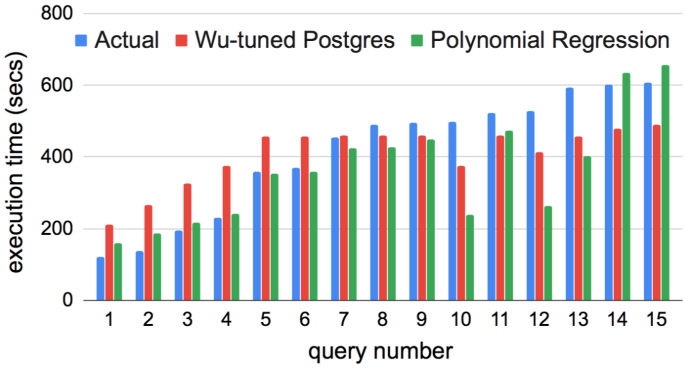
For the three tables, the training curves are shown in Figures 5a, 5c and 5e and their respective predictions are shown in Figures 5b, 5d and 5f respectively. The test queries were picked randomly. As we can see, even a simple ML model (polynomial regressor) trained on very few queries is doing better than the PostgreSQL's model with a mean q-error of 1.2, 1.13 and 1.17 against PostgreSQL's 9.8, 1.96 and 1.36 for $CF = 1, -1$ and 0 respectively. Note that the models were tested on around 30 queries in each case but only 15 (random) have been shown here due to space constraints.



(a) Prediction on test queries for $CF = 1$



(b) Prediction on test queries for $CF = -1$



(c) Prediction on test queries for $CF = 0$

Figure 6: Using models in Figure 5 to predict times of $Q\langle R, movie\_id, v1, v2\rangle$-type test queries

Further, we also did experiments to check if the same model can be used for different values for $v1$, i.e. to inspect if the modeling can be range-location-agnostic (in other words can we predict same time for $[v1, v2]$ or $[v3, v4]$ if both have same cardinality?). To answer this, we used the models created for $v1 = 1$ to answer queries of all values of $v2$ and as expected, the models performed poorly (shown in Figure 6 with q-errors of 1.08, 6.02 and 1.25 respectively). Thus, the models cannot be range-location-agnostic. A possible way to tackle this problem is to bucketise the whole range of cardinality into k buckets and use the same model for each bucket. This is because, for instance, a model for $Q\langle R, movie_id, 1, v2\rangle$ and $Q\langle R, movie_id, 2, v2\rangle$ should not be much different. But, the challenge is how to bucketise the whole range so that the training time is less, the space required to store the model is less and the prediction time is less.

7 Related Work

Wu et al. followed their statistical tuning [8] approach by trying to tackle the noise in the cost tuning parameters [9]. As part of ML-based research, Chetan et al. [3] used random forest technique on an ensemble of models (like k-nearest neighbour) to give a band of execution time. Ganapathi et al. [2] took plan-level features of a query plan and applied a Kernel Canonical Correlation Analysis (KCCA) approach to predict not just time but other metrics like CPU usage, memory usage, etc. Akdere et al. [1] went for a hybrid modeling approach by using SVM at plan level and linear regression at operator level. Li et al. [5] applied linear regression on all the nodes of a plan and scaled the results at runtime to get better estimates. Finally, there has been recent work [6] that learns a neural network model for each operator and combines the result using another neural network on top.

8 Conclusions and Future Work

In conclusion, the cost model of PostgreSQL for index scan does not seem to provide good estimations and it seems possible to leverage the data layout to build a simple and explainable machine learning model which can do better predictions. The next step is to see if fitting piece-wise curve on better training data can improve predictions. Then, the next step is to see how to handle all $Q\langle R, movie_id, v1, v2\rangle$-type queries then head on to handle the equality predicates. After that, other scan operators like bitmap heap scan and internal node operators like join and aggregate need to be looked at. Uncertainties like spilling are also a part of the future aspects of the project.

References

[1] M. Akdere and U. Çetintemel. Learning-based Query Performance Modeling and Prediction. In *ICDE*, 2012.

[2] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, 2009.

[3] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAM*, 2008.

[4] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper and T. Neumann. How Good Are Query Optimizers Really? *PVLDB*, 9(3), 2015.

[5] J. Li, A. C König, V. Narasayya, and S. Chaudhary. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *PVLDB*, 5(11), 2012.

[6] R. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. In arXiv, 2019.

[7] L. F. Mackert and G. M. Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *TODS*, 14(3), 1989.

[8] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *ICDE*, 2013.

[9] W. Wu, Xi Wu, H. Hacigümüs and J. F. Naughton. Uncertainty Aware Query Execution Time Prediction. *PVLDB*, 7(14), 2014.