Improving PostgreSQL Cost Model

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

BY

Pankhuri

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

June, 2015

Declaration of Originality

I, **Pankhuri**, with SR No. **04-04-00-10-41-13-1-10301** hereby declare that the material presented in the thesis titled

Improving PostgreSQL Cost Model

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the year **2014-2015**.
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: Advisor Signature

DEDICATED TO

My Family

for continuous support

Acknowledgements

I am deeply grateful to Prof. Jayant Haritsa for his unmatched guidance, enthusiasm and supervision. He has always been a source of inspiration for me. I have been extremely lucky to work with him.

Also, I am thankful to Prof. Matthew Jacob Thazhuthaveetil and Anshuman Dutt for discussions and suggestions. My sincere thanks goes to my fellow lab mates for all the help and suggestions.

Finally, I am indebted with gratitude to my family for their love and inspiration that no amount of thanks can suffice. This project would not have been possible without their constant support and motivation.

Abstract

Prediction of execution time for different queries is an important task in database field. Cost models of optimizers are used for for this prediction. But the prediction is inaccurate due to two types of errors: (i) selectivity estimation error and (ii) cost-modeling error. There are some existing works for handling selectivity estimation error, like *Plan Bouquet*. But cost-modeling error is still to be handled, so we are focusing on that.

A recent work shows that by adjusting the tunable parameters, cost-modeling errors associated with current optimizers can be reduced significantly. In their experiments, when correct cardinalities were substituted in query plans, mean relative error for a particular set of TPC-H queries due to PostgreSQL cost model was within 47%.

Through our experiments, we found that if we vary the constants associated with selected predicates of some queries, then errors are low for some constants but increase for others. Further experiments indicated that this happens because some static parameters of the cost model, which are hard-coded by the optimizer-designers, need to be tuned before using. Although these errors were not more than 47%, these can be reduced by tuning the static parameters. So in this thesis, we show our work with static parameters to reduce the cost-modeling errors. By tuning these static parameters, errors are significantly reduced over some queries without much increasing errors for other queries.

Moreover, previous work mentions that one of the parameters of PostgreSQL cost model, namely *random_page_cost*, which is the cost of fetching a disk page randomly is difficult to calibrate because pure random access is difficult to achieve and there are some uncertainties in estimations related to this parameter. We designed such a query which could achieve pure random access and we experimentally explored the reasons for poor prediction by this parameter. We found that three parameters associated with *random_page_cost* can cause prediction errors. We experimentally verified that out of these three parameters, one is not erroneous while the other two static parameters are erroneous and it is equally important to correct both of them for proper prediction. We also try to model one of these parameters correctly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Contributions

Prediction of execution time for different queries is an important task in database field. Cost models of optimizers are used for this prediction, but the prediction is inaccurate due to two types of errors: (i) selectivity estimation error and (ii) cost-modeling error. By cost-modeling error, we mean the error obtained after substituting correct cardinalities in the query plan. There are some existing works for handling selectivity estimation error, like *Plan Bouquet*[7]. But cost-modeling error is still to be handled, so a cost model with as low modeling error as possible is required.

We are working with PostgreSQL cost model since it is the most advanced open source database. The optimizer designers have designed a default cost model for PostgreSQL which needs to be tuned according to hardware configurations of the system on which queries have to be executed, before using it. The authors of [8] proposed a method for adjusting the tunable parameters of default PostgreSQL cost model. They calibrated these parameters, taking into account hardware configurations of the system. They showed some experiments, where after this calibration, mean relative error for a particular set of TPC-H queries due to PostgreSQL cost model was within 47%.

Although the prediction errors using the approach of [8] was within 47% for that set of queries, variation in the type of operations allowed in database should be properly taken into account. For example, execution time for TPC-H query 3 involving hash join as a major part of its query plan is predicted well by their cost model, on our system, giving a prediction error of only 10%. But query 13 involving sort is predicted poorly giving error of 184%. On exploring the reason for this behavior, we found that this happens because some static parameters of the cost model, which are hard-coded by the optimizer-designers, need to be tuned before using.

So we calibrated these static parameters to reduce the prediction errors over such queries.

These static parameters can be calibrated and our experimental results show that the cost model obtained after calibrating these parameters work as well as the approach in [8] in most cases and shows significant improvement in some cases. For example, by calibrating the static parameters, prediction error for query 13 is reduced to 42% without any significant increase in error for other queries.

Moreover in [8], authors mention that one of the tunable parameters called *random_page_cost*, which is the cost of fetching a disk page randomly is difficult to calibrate because pure random access is difficult to achieve and there are some uncertainties in estimations related to this parameter. Their cost model does not work well for query plans involving this parameter, like TPC-H query 2. So a more accurate method than the one described by them is required for calibrating this parameter.

We designed such a query which could achieve pure random access and calibrated *random_page_cost*. But while calibrating this parameter, we found that it has many multipliers and some of them are estimated by the optimizer. These estimators are static and calibration of *random_page_cost* alone will not help unless these static parameters are tuned. The authors of [8] also mention that there is uncertainty in some estimations related to *random_page_cost*. We identified three parameters responsible for poor prediction by *random_page_cost* experimentally and through carefully designed experiments, we were able to isolate the effect of these parameters and study them. We found that out of these three parameters, one is not erroneous while the other two are erroneous and it is important to correct both of them for proper cost prediction. We also try to model one of the erroneous parameters correctly.

1.2 Organization

The remainder of this thesis is organized as follows. Chapter 2 gives background on cost model and discusses the approach of [8]. In Chapter 3, we describe cost-modeling errors in detail. In Chapter 4, we discuss the calibration of some specific static parameters of the cost model. In Chapter 5, we discuss about the parameters related to *random_page_cost*. In Chapter 6, we describe a methodology for calibration in an uncontrolled environment. In Chapter 7, we show some experimental results and we conclude in Chapter 8.

Chapter 2

Background

2.1 PostgreSQL Cost Model

In general, whenever a query is given to the database engine, it goes to the optimizer which explores various plans and finds the optimal plan using a Cost Model. The optimal plan is given to the executor which executes the query using this plan. This is illustrated in Figure 2.1.

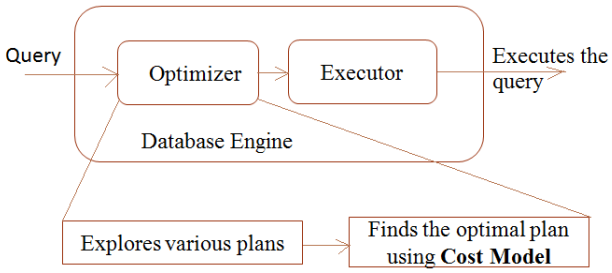


Figure 2.1: Execution of a query

Let us denote the cost of a plan by C_{plan}. The optimizer explores various plans for a query and the one with minimum C_{plan} is chosen as the optimal plan. We are looking at cost model from the perspective that besides helping in selecting the optimal plan, C_{plan} can also be used to predict the execution time of a query. We try to explain the calculation of C_{plan} for a query plan through Figure 2.2.

It shows the plan tree for TPC-H query 13. There are many nodes in the tree and cost for each node k is calculated as $C_k = n_k^T c$. Here C_k is the cost contributed by node k, n_k is the cardinality vector for node k and c is the cost vector for the respective cost model. Then cost

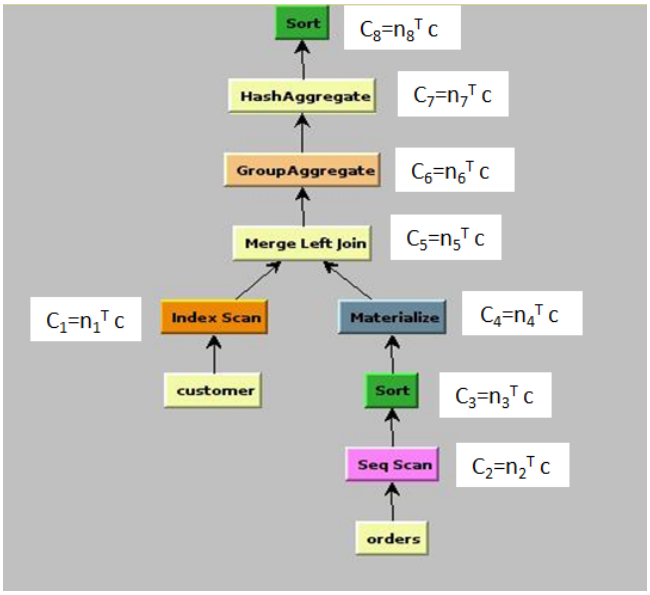


Figure 2.2: Plan Tree for TPC-H query 13

of the whole plan, C_{plan} is calculated as: $C_{plan} = \sum_{k=1}^{n} C_k$, where n is the total number of nodes in the query plan. In Figure 2.2, n=8.

In this thesis, we are dealing with PostgreSQL cost model since it is the most advanced open source database. PostgreSQL cost model comprises of five cost parameters[1]:

1. ***seq_page_cost***(c_s): the cost of a sequential disk page fetch
2. ***random_page_cost*** (c_r): the cost of fetching a disk page randomly
3. ***cpu_tuple_cost*** (c_t): the cost of processing a tuple
4. ***cpu_index_tuple_cost***(c_i): the cost of processing an index entry during an index scan
5. ***cpu_operator_cost***(c_o): the cost of performing an operation

The cost of an operator, C_o is calculated as [8]:

$$C_o = n_s.c_s + n_r.c_r + n_t.c_t + n_i.c_i + n_o.c_o \tag{2.1}$$

where

n_s: number of disk pages fetched sequentially

n_r: number of disk pages fetched randomly

n_t: number of tuples processed

n_i: number of index entries processed during an index scan

n_o: number of operations performed

The estimated cost of a query plan,C_{plan} is given as :

$$C_{plan} = \sum_{i=1}^{n} C_{oi}, \tag{2.2}$$

where C_{oi} is the cost of i^{th} operator in the query plan having n operators. Let us represent the cost vector as $c = [c_s, c_r, c_t, c_i, c_o]^T$ and the cardinality vector as $n = [n_s, n_r, n_t, n_i, n_o]^T$.

We are using C_{plan} for predicting execution time of the query. Errors in this prediction by C_{plan} may be due to: (i) selectivity estimation error and/or (ii) cost-modeling error. Here we are concerned only with the cost-modeling error. All the errors in prediction by C_{plan} after substituting true cardinalities in the plan are cost-modeling errors. By true cardinality substitution, we mean correct substitution of number of tuples in the query plan.

2.2 Problem Definition

Let C be a given cost model and Q be the set of all queries of a given workload on a database D. Let
c_i be the cost of plan,
t_i be the actual execution time of query $q_i \in Q$ and
N be the total number of queries in Q.
Authors in [8] define an error metric called "Mean Relative Error", which we also use:

$$MRE = \frac{1}{N} \sum_{i=1}^{N} |\frac{c_i - t_i}{t_i}| \tag{2.3}$$

In default PostgreSQL cost model, c_i and t_i can't be mapped to one another, so like in [8], we also use linear regression for calculating MRE. In the modified cost models which we will be discussing throughout the thesis, c_i is calibrated in units of time. So c_i and t_i are in same units and thus, MRE can be calculated.

For an individual query $q_i \in Q$, *Relative Error* is defined as:

$$RE = \frac{c_i - t_i}{t_i} \tag{2.4}$$

MRE is mean over absolute values of the REs of the queries in Q, and absolute values are taken in order to avoid cancellation of REs for different queries. MRE is always non-negative. RE

can be positive or negative, where positive RE means overestimation and negative RE means underestimation of execution time by the cost model.

DEFINITION 1:(PROBLEM DEFINITION) *For a set of queries Q on a database D, we are concerned with the problem of improving the cost model of PostgreSQL by reducing its errors. We aim to improve it in such a way that MRE is as low as possible.*

MRE is calculated after correct cardinality substitution. If there are two cost models C1 and C2 with mean relative errors MRE1 and MRE2 respectively, over a given Q and D, then the one with lower MRE is a better cost model, i.e., C1 is better than C2, if $MRE1 < MRE2$. If MREs for two cost models are same, the one with lower value for maximum RE over a given Q and D is better.

The authors in [8] were only dealing with MRE. We are also dealing with MRE for a set of queries, but we mention *Relative Error* as well for individual queries.

2.3 Existing Approach for Reducing Cost-Modeling Errors

This section describes the work done in [8] where the authors design a complete, concise and simple set of five calibration queries for the five PostgreSQL cost parameters. They introduce each cost parameter one by one for building up the five queries:

1. ***cpu_tuple_cost***: The calibration query is of the form:

   $$select * from\ R$$

   Here R is a buffer pool resident relation. No I/O cost is involved and only c_t plays role in cost calculation :

   $$n_t.c_t = t_1$$

   t_1 is the execution time of this calibration query and $n_t = |R|$. Thus, c_t can be calculated easily.

2. ***cpu_operator_cost***: The calibration query is of the form:

   $$select\ count(*)\ from\ R$$

   Here also, R is buffer pool resident. No I/O cost is involved and only c_t, c_o play role in cost calculation :

   $$n_t.c_t + n_o.c_o = t_2$$

   Here, $n_t = n_o = |R|$ and t_2 is the query execution time. Having calculated c_t from above, c_o can be computed.

3. ***cpu_index_tuple_cost***: The calibration query is of the form:

$$select * from\ R\ where\ R.attr < a$$

Here R is buffer pool resident and so, no I/O cost is involved. *attr* is an attribute of R on which a clustered index is built and a is picked so that an index scan is chosen by the optimizer. The cost is calculated as:

$$n_t.c_t + n_o.c_o + n_i.c_i = t_3$$

t_3 is query execution time and $n_t = n_o = n_i$ =Number of tuples returned by the query. c_i is the only unknown in this equation.

4. ***seq_page_cost***: The calibration query is of the form:

$$select * from\ R$$

Here R is not buffer pool resident and is scanned sequentially. The cost is calculated as:

$$n_t.c_t + n_s.c_s = t_4$$

t_4 is query execution time and the only unknown c_s can be calculated easily.

5. ***random_page_cost***: The calibration query is of the form:

$$select * from\ R\ where\ R.un\_attr < b$$

Here R is not buffer pool resident and *un_attr* is an attribute of R on which an unclustered index is built. b is chosen so that an index scan is chosen by the optimizer. According to [8], it is difficult to get pure random access and some local sequential accesses are also included. So all the five cost parameters play a role:

$$n_t.c_t + n_o.c_o + n_i.c_i + n_s.c_s + n_r.c_r = t_5$$

Here, t_5 is the query execution time and having calculated the four cost parameters above, the only unknown c_r can be calculated.

Multiple queries of each type were executed for more robustness. This procedure was used in [8] and we repeated it for our experiments. We refer to this cost model as $Calib_{tunable}$ *model*.

2.3.1 Calibrated Values of the Cost Parameters

The values of the five calibrated parameters of the cost model, obtained on our system through the approach of [8], are shown in Table 2.1. Column 2 shows the values of default cost parameters, while Column 3 shows the values of parameters obtained through the approach of [8] on our system. In Column 4, we show the ratios among the cost parameters obtained through approach of [8] (normalized with respect to *seq_page_cost*) for our hardware configuration, which is significantly different from the ratios among the default parameters. We can see that $c_r/c_s = 4.0$ according to the default cost model, but $c_r/c_s = 2.6$ for the calibrated parameters. So random page fetch is not as costly as expected by the optimizer designers for our system, thus affecting the choice of plans.

Table 2.1: Values of PostgreSQL Cost Parameters

Cost Parameters	**Default Values**	**Values of Calibrated Parameters**	**Ratios among the Calibrated Parameters**
c_s	1.00	1.05e-1	1.00
c_r	4.00	2.73e-1	2.60
c_t	1.00e-2	2.88e-4	2.74e-3
c_i	5.00e-3	6.00e-5	5.72e-4
c_o	2.50e-3	2.13e-4	2.03e-3

2.3.2 Prediction Errors on TPC-H Benchmark Queries

In Table 2.2, we show the relative errors in prediction of query execution time by the cost model on 20[1] out of 22 TPC-H benchmark queries. Column 2 shows relative errors with the default parameters after applying linear regression. Column 3 shows the errors with calibrated parameters obtained using the approach of [8] on our system. Err_{def} and Err_{calib} denote the relative error with default cost parameters and calibrated cost parameters, respectively. With both the cost models, in most of the queries, cost is overestimating time, while for few queries it is underestimating. Overestimation is shown by positive errors and underestimation is shown by negative errors.

It is seen that for queries 2, 13 and 21, default cost model performs much better than the calibrated cost model. But on an average, it predicts execution time poorly as can be seen from the much higher MRE for default model as compared to the calibrated model. This happens even on applying linear regression, since the ratios among the default parameters is not correct for our system as can be seen from Table 2.1.

Although the errors with the default model are always within 100% for this set of query, but since we are applying linear regression, if there is poor mapping between cost and time even for a single query, errors for other queries in the set will also increase. So high error in one query can affect other queries, but this is not the case with calibrated model. Moreover, MRE with the calibrated model is significantly lower than the default model.

The above mentioned five cost parameters are set in terms of *seq_page_cost* in the default cost model. These are set by the optimizer designers and the model is meant for selecting the optimal plan out of the various possible plans for a query. In [8], the authors show that the same cost model can also be used to predict execution time of a query if hardware configuration of

[1]Query 15 is excluded because it includes view and we have not worked on view yet.
Query 20 is excluded because cardinality substitution was difficult for it.

Table 2.2: Errors for TPC-H Benchmark Queries

Query	$\boldsymbol{Err_{\text{def}}}$	$\boldsymbol{Err_{\text{calib}}}$
1	0.95	-0.65
2	0.69	0.75
3	0.85	0.10
4	0.81	0.26
5	0.82	0.23
6	0.81	0.33
7	0.85	0.08
8	0.85	0.01
9	0.80	0.41
10	0.84	0.12
11	0.78	0.57
12	0.81	0.28
13	0.59	1.84
14	0.85	0.08
16	0.91	-0.35
17	0.86	0.03
18	0.88	-0.15
19	0.85	0.06
21	0.69	1.10
22	0.78	0.43
MRE	**0.81**	**0.40**

the system on which queries have to be executed are taken into account. They set these tunable cost parameters in terms of execution time, rather than in terms of *seq_page_cost*.

This cost model works well for queries involving particular operations, like for hash in TPC-H query 3 but may work poorly for some other queries involving other operations, like for sort in TPC-H query 13. So we try to further improve the cost model by reducing its cost-modeling errors, where we also take static parameters into account.

Chapter 3

Cost-Modeling Errors

The errors in prediction of execution time by cost model may be due to errors in: (i) selectivity estimation and/or (ii) cost-modeling. Selectivity estimation error is handled by us by explicitly substituting correct cardinalities in the plan. Cost-modeling errors can be due to: (i) tunable parameters of the cost model and/or (ii) static parameters of the cost model. We define cost-modeling error as:

$$Cost - modeling \quad error = func(tunable\_parameters, static\_parameters) \tag{3.1}$$

Thus cost-modeling error is a function of tunable and static parameters of the cost model. In this thesis, we are considered with the cost-modeling errors of PostgreSQL only. Tunable parameters are the five cost parameters of PostgreSQL: c_s, c_r, c_t, c_o and c_i. These have been given some specific values by the optimizer designers, but are subject to change depending on the hardware requirements.

Static parameters are hard-coded by the optimizer designers and can be of many types. Some examples are given below:

1. Multipliers: In different operations, different multipliers of tunable parameters are used while calculating the cost. In some cases, these operator-specific multipliers need to be tuned. This is the case with sort, as described later.

2. Estimation: In some cases, estimations are made by PostgreSQL while calculating cost, for example, number of pages to be accessed are estimated while calculating the cost of an index scan. These estimations are made using various functions and are erroneous many times.

3.1 Motivation for Calibrating Static Parameters

Using the approach of calibration suggested by [8], prediction errors for many queries reduced significantly as compared to the default model. But let us consider TPC-H query 3 in which we vary the selectivities over some chosen predicates. We observe that, at selectivity of 5%, relative error was 3%. But at selectivity 95%, error increased to 39%. We found that this error could not be handled by calibration of tunable parameters alone. There are various types of operations in PostgreSQL and c_o is calibrated according to only one of these. So some static parameters need to be tuned for other operations. Sort is one such operation. At selectivity 5%, sort had a little contribution in cost of the plan, but at selectivity 95%, this contribution increased significantly, increasing the error. By calibrating the static parameters, the error at 95% selectivity was reduced to 10%, which is further explained in Section 7.2.2. Thus, calibration of static parameters is necessary for reducing cost-modeling errors of PostgreSQL.

Out of the five cost parameters in PostgreSQL cost model, three of them: c_s, c_t and c_i are used limited number of times. The calibration queries used by [8], corresponding to these three parameters are similar to their use cases, and multipliers or estimations associated with them are generally correct. For example, c_s is only used in case of sequential scan where it is multiplied by the number of pages scanned sequentially, which is correctly estimated by the optimizer. The multiplier of c_t is number of tuples processed, which we are ensuring is correct by correct cardinality substitution. c_i is used only in case of index scan and its multiplier is also made correct through correct cardinality substitution. So the remaining two parameters are of primary importance when calibrating the static parameters: c_o and c_r. The static parameters related to these two cost parameters are discussed in the following two chapters.

Chapter 4

Calibrating Static Parameters Related to *cpu_operator_cost*

We discuss the calibration of static parameters related to c_o in this chapter.

PostgreSQL uses different formulas for calculating cost of different operations. We worked with these formulas and tried to calibrate the multipliers of c_o in them. These multipliers are static parameters. Let us refer to the multipliers as m, which will be different for different operators. Then we define $m.c_o$ as static parameter for that operator. We follow the following approach for calibrating static parameters for different operators:

- **Sort** : As explained in Chapter 1, the cost model designed by [8] does not work well for queries involving sort. So we calibrate the static parameter for sort.

  We calibrate static parameter only for quicksort and it has been verified experimentally that it works well for all types of sort, except for disk-based sort as it involves *random_page_cost*. For this, we use calibration query of the form:

  $$select * from\ R\ order\ by\ R.attr1$$

  Here, R is a buffer pool resident relation, so no I/O cost is involved. Sequential scan and quicksort are forced here. Quicksort is forced by increasing the *work_mem*. The estimated cost of this query is calculated as:

  $$c_t.n_t + 2m.c_o.T.log(T) + m.c_o.T = t_6$$

  t_6 is the query execution time and T is the number of tuples sorted. Here, $n_t = T = |R|$. As discussed above, m is the multiplier of c_o and $m.c_o$ is the desired static parameter. Having the c_t value as calculated in Section 2.3, $m.c_o$ is the only unknown and can be calculated easily. We refer to this static parameter as m_o^{sort}.

- **Nested Loop Join** : We calibrate a static parameter for nested loop join and we currently consider the case of queries involving nested loop join in which both the inner and outer children are sequentially scanned. The calibration query is of the following form after forcing nested loop join:

  $$select * from\ R1,\ R2\ where\ R1.A{=}R2.B$$

  The inner relation being scanned sequentially is materialized and cost of rescan of inner relation is: $cost\_rescan = I.m.c_o$. Thus, the cost for whole query is calculated as:

  $$O.cost\_rescan + O.I.c_t + C1 + C2 = t_7$$

  Here, t_7 is the query execution time and the relations R1, R2 are not buffer pool resident.
  O: Number of rows in outer child
  I: Number of rows in inner child
  C1: Cost of sequential scan on R1
  C2: Cost of sequential scan on R2
  c_t is computed in Section 2.3 and hence, $m.c_o$ is the only unknown here. We refer to this static parameter as m_o^{nest}.

- **Generic** : This static parameter is generic and is defined for all operations other than sort and nested loop join described above, like hash and aggregation. c_o for this case is calibrated for count() as in [8] and shown in Section 2.3. This c_o value has been confirmed experimentally to work well for hash and aggregation operations. Thus, m=1 and no calibration of any static parameter was required in this case as c_o was working well for these operations. We can say that static parameter= c_o in this case. We refer to this as $m_o^{generic}$.

We worked with multiple queries for calibrating static parameter related to each type of operation. We refer to the model obtained by us after calibrating the static parameters as $Calib_{static}$ *model*. Thus, the $Calib_{static}$ *model* consists of five tunable parameters :*seq_page_cost*, *random_page _cost*, *cpu_tuple_cost*, *cpu_operator_cost* and *cpu_index_tuple_cost*. It also consists of three static parameters:
$\mathbf{m_o^{sort}}$: static parameter for Sort operation,
$\mathbf{m_o^{nest}}$: static parameter for Nested Loop Join (involving both sequentially scanned children) and
$\mathbf{m_o^{generic}}$: static parameter for the Generic cases, where $m_o^{generic} = c_o$.

Chapter 5

random_page_cost

We have observed that queries involving c_r are difficult to predict by the cost model. Also the authors in [8] mention that c_r needs a better calibration than the approach proposed by them. So we study c_r in detail and present our observations in this section. We found that some static parameters are related to c_r which must be tuned for better prediction by the cost model and calibrating c_r alone will not help.

The authors of [8] mention that it is difficult to achieve pure random access, but we designed a query that could achieve pure random access and used it to calibrate c_r. We found three parameters which can be responsible for poor prediction by c_r and we designed experiments to isolate the impact of each one of them from the other two. Thus, we studied the impact of these parameters on cost calculation and found that one of the parameters is not erroneous, while the other two are equally important for predicting the cost properly.

This section is organized as follows. In Section 5.1, we discuss our approach for calibrating c_r and in Section 5.2, we discuss about the cost calculation by PostgreSQL for queries involving random page access. In Section 5.3, we enumerate the parameters related to c_r which may cause error in cost calculation. In further sections, we discuss about each parameter in detail.

5.1 Our Approach for calibrating *random_page_cost*

The authors of [8] mention that it is difficult to calibrate c_r because achieving pure random access is difficult and local sequential accesses are unavoidable. But we designed a query that could achieve pure random access and is of form:

$$select\ *\ from\ R\ where\ R.attr{=}a_1\ or\ R.attr{=}a_2\ or\ ...or\ R.attr{=}a_n$$

Here, R.attr is an attribute of R on which an unclustered index is built and R is not memory resident. The values of a_1, a_2,..., a_n are chosen such that each of these is on a different data page.

Let the tuple of R corresponding to R.attr=a_1 be on page p_1, tuple corresponding to R.attr=a_2 be on page p_2,..., tuple corresponding to R.attr=a_n be on page p_n. Our idea is to choose a_1, a_2,..., a_n such that p_1, p_2,..., p_n are different pages, i.e., $p_i \neq p_j$, if $i \neq j, \forall i, j \in \{1, ..., n\}$.

Now, depending upon the values of a_1, a_2,..., a_n, we can control the locations of p_1, p_2,..., p_n on disk. For example, we can create a query in which we will choose the values of a_is such that each p_i is 1000 pages away from the other. In such a way, we can create different queries and experiment for different disk locations.

We did this experiment for different queries where we picked values of a_is in such a way that we could maintain desired distance between the pages accessed. By distance between the pages accessed, we mean difference between their page-identifiers. These are the number of pages as measured by us while query execution and page-identifiers were noted from the file descriptor of PostgreSQL. Table 5.1 shows the number of pages accessed and distance between the pages for different queries. We measure query execution time and then c_r is calculated from the following equation:

$$n_t.c_t + n_o.c_o + n_i.c_i + n_s.c_s + n_r.c_r = time$$

Here,
$n_t = n_i = n_o =$ No. of tuples returned by the query,
$n_s =$No. of index pages accessed by the query,
$n_r =$No. of data pages accessed by the query.
c_t, c_o, c_i, c_s are same as defined earlier and their values are already known.

In this query, we could achieve pure random access of the data pages, which is verified by the fact that number of data pages accessed by this query is equal to the number of tuples returned by the query and all the data pages are far from each other.

Table 5.1 shows the values of c_r obtained for different distances between the pages. By distance, we mean the minimum distance between the pages.

Table 5.1: Values of *random_page_cost* for different distances between the pages

No. of Data Pages Fetched	**Distance**	**% of data pages scanned**	$\mathbf{c_r}$ (ms)
60	10000	0.01	44.90
400	1000	0.07	37.29
6000	100	1.00	36.31
10000	60	1.67	43.58

The mean of the different values of c_r obtained can be used as calibrated value for c_r.

This c_r value is actually the time taken to fetch a page randomly from disk and we call it as *random_disk_access_time*.

5.2 Cost Calculation in PostgreSQL for queries involving random page access

There is a parameter called *correlation* which is used by PostgreSQL. *Correlation* indicates the statistical correlation between physical row ordering and logical ordering of the column values and ranges from -1 to +1[5]. Value of *correlation* close to 0 indicates lack of correlation and value close to ±1 indicates good correlation.

Let us consider the case of a simple query where index scan is done over a relation. In perfectly uncorrelated case and in absence of any buffer, a new data page would have to be fetched for each new tuple. Total no. of data pages that have to be accessed in this case=*N1*, say.

Mackert-Lohman formula takes the effect of buffer, b into account and reduces the no. of data pages that have to be accessed from *N1* to *N2*, say. In other words, *Mackert-Lohman formula* takes *N1* and b as input, applies a function M over them and produces *N2* as output. Lets refer to *N2* as *Mackert_pages*.

$$N2 = M(N1, b) \tag{5.1}$$

Thus, maximum number of data pages that can be accessed randomly is equal to *Mackert_pages*.

Now, let us discuss about the minimum number of pages that can be accessed by a query. Let us assume there are 1000 pages in a relation and each page has a certain number of tuples, and we want to access one-tenth of the total number of tuples present in the entire relation. Then according to the *pigeonhole principle*[2], at least one-tenth of the total number of pages in the relation, i.e., 1/10 * 1000 = 100 pages have to be accessed. PostgreSQL does the same estimation for the minimum number of pages that have to be accessed in a perfectly correlated case, i.e., product of selectivity and size of the relation. Let us refer to these minimum number of pages as *min_pages* and denote it by N'. In a perfectly correlated case, these pages will be accessed sequentially.

For partially-correlated cases, PostgreSQL uses a function of *correlation* to interpolate between the cost estimates for perfectly correlated and perfectly uncorrelated cases. This is done as:

$$N2 * (1 - (correlation)^2) * c_r + N' * (correlation)^2 * c_s \tag{5.2}$$

In general, when a query is executed, all the pages are not accessed randomly, rather some sequential accesses are also there if nearby tuples are brought from nearby data pages. This locality of reference is captured through the use of *correlation* by PostgreSQL.

Let us look at Equation (5.2) in a different way. Let $N3^{random}$ and $N3^{seq}$ denote the number of pages randomly and sequentially accessed during execution of a query, as estimated by PostgreSQL, respectively. In Equation (5.2), $N3^{random}$ and $N3^{seq}$ are the multipliers of c_r and c_s, respectively:

$$n_r = N3^{random} = N2 * (1 - (correlation)^2) \tag{5.3}$$

$$n_s = N3^{seq} = N` * (correlation)^2 \tag{5.4}$$

We can say that $N3^{random}$ is calculated by PostgreSQL after applying some function of *correlation* and *N2*, and $N3^{seq}$ is calculated after applying some function of *correlation* and $N`$.

$$N3^{random} = func1(N2, correlation) \tag{5.5}$$

$$N3^{seq} = func2(N`, correlation) \tag{5.6}$$

There is one more aspect related to c_r. Optimizer designers assume 90% *cache_hit_rate* for the queries being executed. If *random_disk_access_time* is the time taken to fetch a page randomly from the disk, then c_r is defined as:

$$c_r = random\_disk\_access\_time * cache\_miss\_rate,$$

or,

$$c_r = random\_disk\_access\_time * 0.10$$

Let us denote *random_disk_access_time* as C_R. So,

$$c_r = C_R * 0.10 \tag{5.7}$$

As discussed earlier, for cost calculation, c_r is multiplied by the number of pages randomly accessed:

$$N3^{random} * c_r$$

or,

$$N3^{random} * C_R * 0.10 \tag{5.8}$$

This is summarized in part (a) of Figure 5.3.

5.3 Static Parameters related to *random_page_cost*

As discussed in the previous section and in Equation (5.8), while calculating the cost of a query, c_r is multiplied by number of *pages_fetched*, i.e., $N3^{random}$ and c_r is itself a product of C_R and *cache_miss_rate*. So error in predicting execution time of a query by c_r can be due to error in either one or more of the following parameters:

1. *random_disk_access_time*
2. *pages_fetched*
3. *locality of reference*

Here, *pages_fetched* means $N3^{random}$, and *cache_hit_rate* is included in *locality of reference*.

In Section 5.1, we designed queries that could isolate the impact of *random_disk_access_time* from *pages_fetched* and *locality of reference*. There is no effect of *locality of reference* and estimation of number of *pages_fetched* in those queries. If many tuples are accessed from a relation, effect of *locality of reference* cannot be avoided. But since we are accessing few tuples and ensuring that contiguous pages are not accessed, there is no effect of *locality of reference*. As we are measuring and using the correct number of pages accessed by the query, errors of estimation in *pages_fetched* are also avoided. Thus, this query isolates C_R from the other two parameters related to c_r. We observed that C_R does not vary much for different disk locations implying that this parameter is not erroneous in cost calculation for queries involving c_r.

So out of the three factors responsible for prediction error due to c_r in cost of a plan: (i) *random_disk_access_time* (ii) *pages_fetched* and (iii) *locality of reference*, *random_disk_access_time* does not vary much and hence, is not a source of error in most cases.

Thus, the following static parameters related to c_r are responsible for poor prediction:

1. *pages_fetched*
2. *locality of reference*

We have studied both of these parameters separately, as described in the following sections.

5.4 *Pages_fetched*

In this section, we describe about some queries designed by us where we tried to isolate the impact of number of *pages_fetched* from the other two parameters. After this, there were still

errors in cost prediction implying that number of *pages_fetched* is erroneous in many queries and needs to be corrected for proper cost prediction.

We also show some experimental observations which illustrate the fact that both *pages_fetched* and *locality of reference* are equally important for correct cost prediction. Even if one is erroneous, we can get high errors.

5.4.1 Error in number of *pages_fetched*

In this subsection, we show the impact of error in number of *pages_fetched* in some specially designed queries. We use queries of the form:

$$select\ *\ from\ R\ where\ R.attr{<}a$$

Here index scan is forced on relation R and $R.attr$ is some chosen attribute of R. We experiment with many queries of this form by selecting different $R.attr$ and a values.

We measure execution time of these queries and define four types of cost here: (i) $Cost1_{pages\_est}$ is the cost of a query when we substitute correct cardinalities in the query, but number of *pages_fetched* is left to the optimizer to estimate and the value of c_r used is that calculated by the approach of [8]. (ii) $Cost2_{pages\_est}$ is same as $Cost1_{pages\_est}$ except that the value of c_r used is substituted after ensuring correct *locality of reference*. The details of finding this c_r which ensures correct *locality of reference* is discussed in Section 5.5. (iii) $Cost1_{pages\_corr}$ is the cost of a query when along with correct cardinalities, we also substitute correct number of *pages_fetched* in the query plan and the value of c_r used is that calculated by the approach of [8]. (iv) $Cost2_{pages\_corr}$ is same as $Cost1_{pages\_corr}$ except that the value of c_r used is substituted after ensuring correct *locality of reference*.

The values of remaining four cost parameters, except for c_r are same in all these four types of costs and are the same as calculated using the approach of [8] and shown in Section 2.3.

When we say we substituted correct number of *pages_fetched*, we mean we substituted only correct number of total pages accessed. We were not able to substitute correct values of $N3^{random}$ and $N3^{seq}$ since currently there is no way of correctly estimating it, but substituting correct total number of *pages_fetched* gives better estimation of *pages_fetched* than the one used by PostgreSQL. So, although we could not completely remove, but we have reduced the error due to incorrect estimation in the number of *pages_fetched*.

$Error1_{pages\_est}$, $Error2_{pages\_est}$, $Error1_{pages\_corr}$ and $Error2_{pages\_corr}$ are the relative errors of $Cost1_{pages\_est}$, $Cost2_{pages\_est}$, $Cost1_{pages\_corr}$ and $Cost2_{pages\_corr}$, respectively with respect to execution time.

The four different types of errors are illustrated in Figure 5.1. $Error1_{pages\_est}$ shows the case where neither *pages_fetched* nor *locality of reference* is correct. This is shown in Column

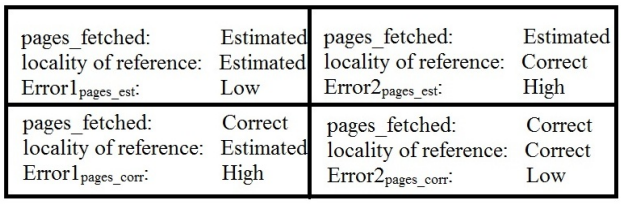


Figure 5.1: Four different types of errors

6 of Table 5.2 and in this case, we get low errors because multiple errors are canceling each other. Error in number of *pages_fetched* makes overestimating prediction error in these queries, while error in *locality of reference* makes underestimating prediction error, thus canceling each others effect when both are present. $Error2_{pages\_corr}$ shows the case when both number of *pages_fetched* and *locality of reference* are correct. These errors are low as can be seen from Column 7 of Table 5.3. As discussed in next Section 5.5, this is the best prediction possible by the cost model, but it is shown just to illustrate that both both *pages_fetched* and *locality of reference* need to be correct for proper prediction of queries involving random page access. If even one is erroneous, errors can be high. Both $Error1_{pages\_est}$ and $Error2_{pages\_est}$ are low for the bottom rows of Tables 5.2 and 5.3 because estimation of the number of *pages_fetched* by the optimizer is good for those queries.

Table 5.2: Errors with estimated number of *pages_fetched*

R.attr	***a***	**Time (s)**	***Cost1* (s)**	***Cost2* (s)**	$\mathbf{Error1_{pages\_est}}$	$\mathbf{Error2_{pages\_est}}$
ss_store_sk	10	516.7	716.0	3913.2	0.386	6.573
ss_store_sk	40	1815.2	2493.7	13628.5	0.374	6.508
cs_call_center_sk	2	102.0	375.4	607.1	2.680	4.952
cs_call_center_sk	16	532.6	1948.1	3150.6	2.657	4.915
cs_catalog_page_sk	1000	47.8	33.4	41.9	-0.301	-0.123
cs_catalog_page_sk	8000	205.6	150.2	188.2	-0.269	-0.085
ss_ticket_number	100000	4.6	5.0	4.7	0.075	0.003
ss_ticket_number	1500000	69.3	75.2	70.2	0.084	0.011

It is difficult to measure the correct number of *pages_fetched* by each relation in benchmark queries and hence, experiments with those queries after substituting the correct number of *pages_fetched* are not done.

Our conclusion is that number of *pages_fetched* is erroneous in many queries and needs to be corrected for proper cost prediction. One more observation is that both number of *pages_fetched* and *locality of reference* are equally important for proper cost prediction.

Table 5.3: Errors with correct number of *pages_fetched*

R.attr	***a***	**Time (s)**	***Cost1* (s)**	***Cost2* (s)**	$\mathbf{Error1_{pages\_corr}}$	$\mathbf{Error2_{pages\_corr}}$
ss_store_sk	10	516.7	95.7	516.2	-0.811	-0.001
ss_store_sk	40	1815.2	334.7	1804.7	-0.810	-0.006
cs_call_center_sk	2	102.0	63.2	101.7	-0.381	-0.002
cs_call_center_sk	16	532.6	332.4	534.7	-0.370	0.004
cs_catalog_page_sk	1000	47.8	37.8	48.0	-0.210	0.004
cs_catalog_page_sk	8000	205.6	167.5	212.4	-0.180	0.030
ss_ticket_number	100000	4.6	5.3	4.6	0.080	0.010
ss_ticket_number	1500000	69.3	75.4	70.3	0.090	0.010

5.4.2 Modeling the function of *correlation*

As discussed in Section 5.2, $N3^{random}$ is calculated by PostgreSQL after applying some function of *correlation* and *Mackert_pages*, i.e., *N2*. $N3^{seq}$ is calculated after applying some function of *correlation* and *min_pages*, i.e., *N'*.

$$N3^{random} = func1(N2, correlation) \tag{5.9}$$

$$N3^{seq} = func2(N', correlation) \tag{5.10}$$

In order to model appropriate functions for estimating $N3^{random}$ and $N3^{seq}$, we performed some experiments which are shown in this subsection.

It is difficult to determine the number of pages accessed randomly and sequentially accessed while executing a query, i.e., $N3^{random}$ and $N3^{seq}$ respectively, so we followed a heuristic to get an approximation of these numbers. By making some changes in the PostgreSQL code, we noted the time interval to access each page and we also measured the distinct number of pages accessed while executing each query. We observed that in perfectly correlated case where all pages are sequentially accessed, all pages were accessed in microseconds and from queries in Section 5.1, we can see that pages accessed randomly take milliseconds. So we assumed the pages for which access time was greater than a millisecond to be randomly accessed, i.e., $N3^{random}$. We ensured that the buffer cache is greater than the size of relation being used, thus, every page was accessed maximum once. So each distinct page was either sequentially or randomly accessed and hence, subtracting $N3^{random}$ from the distinct number of pages gives $N3^{seq}$.

The experiment performed by us is discussed here. We create a relation R and its columns are generated by us corresponding to different *correlation* values. Then we execute queries of

form:

$$select\ *\ from\ R\ where\ R.attr<a$$

R.attr are different columns of R corresponding to different *correlation* values and index scan is forced on R. We execute these queries for *correlation* values in the ranges -0.89 to -0.30 and +0.10 to +0.89. Values of a were chosen for varying selectivities and we measure *Mackert_pages*, *min_pages*, $N3^{random}$ and $N3^{seq}$ for each query we execute.

Generating two orderings having desired *correlation* is a difficult task and so, we had limited data to model the functions for $N3^{random}$ and $N3^{seq}$. However, we modeled some crude functions using some of the data we had and we tested these functions over the remaining data generated by us. These functions are:

$$N3^{random} = N2 - 242 * (correlation)^2 - N2 * correlation + 343 * correlation \tag{5.11}$$

$$N3^{seq} = (N`)^2 - 2050 * (correlation)^2 + N` * correlation - 38 * N` + 2210 * correlation \tag{5.12}$$

These functions give better estimates of number of pages randomly and sequentially accessed, respectively as compared to the functions used by PostgreSQL. Some of the observed and estimated values for $N3^{random}$ are shown in Table 5.4. The observed values are shown in Column 2, and the values estimated using the function of PostgreSQL are shown in Column 3. Column 4 shows the values estimated by the function modeled by us. Root-mean-square errors using the functions modeled by us are lower than the functions used by PostgreSQL. These functions can be modeled better if more data is available.

Table 5.4: Observed and estimated values of number of pages randomly accessed

Correlation	**Observed N3^{random}**	**Estimated N3^{random} by PostgreSQL**	**Estimated N3^{random} by Equation (5.11)**
0.48	2720	1000	2619
0.09	4539	264	4414
0.89	222	773	639
0.29	3937	508	3531
0.68	1244	1343	1658

The functions plotted in this way would model the estimation of number of *pages_fetched* while executing the query. This model will also estimate *locality of reference* at higher granularity since it is a function of *correlation*.

The functions used by PostgreSQL for estimating $N3^{random}$ and $N3^{seq}$ are not actually meant to estimate number of pages randomly and sequentially accessed respectively and hence, gives high errors in Table 5.4. PostgreSQL uses these functions just to interpolate between maximum and minimum costs for a partially correlated case. Since we are interested in using the cost model for predicting query execution time, we are modeling the functions for $N3^{random}$ and $N3^{seq}$ such that they estimate number of pages randomly and sequentially accessed respectively.

5.5 *Locality of reference*

By *locality of reference*, we mean two things:

1. *locality of reference* at a higher granularity which is captured by PostgreSQL using *correlation*. *Correlation* accounts for sequential accesses for the case where nearby tuples are fetched from nearby pages on disk.

2. *locality of reference* at a lower granularity, i.e., *cache_hit_rate*. It captures the case where nearby tuples are fetched from pages already in the buffer cache.

5.5.1 Error in *locality of reference*

Lets consider queries of the following form:

$$select\ *\ from\ R\ where\ R.attr < a$$

where index scan is forced on R. We performed some experiments with these queries to calibrate c_r. *R.attr* were chosen to correspond to different *correlation* values. We have measured the total number of *pages_fetched* in these queries and substitute it while calculating the cost of query execution. Because of the reasons mentioned in Section 5.4.1, we were not able to substitute correct values of $N3^{random}$ and $N3^{seq}$, but substituting correct total number of *pages_fetched* gives better estimation of *pages_fetched* than the one used by PostgreSQL. So, although we could not completely remove, but we have reduced the error due to incorrect estimation in the number of *pages_fetched*. Some error due to incorrect function of *correlation*, which again indicates locality of reference at a higher granularity as discussed in Section 5.2 is still there. In the Section 5.1, we verified that *random_disk_access_time* is not erroneous. Thus we are able to isolate the effect of *locality of reference* from the other static parameters related to c_r. Here, by *locality of reference*, we mean both *cache_hit_rate* and the locality of reference at higher granularity which is captured by *correlation*.

The calibration queries were selected such that *R.attr* correspond to different columns of *store_sales* and *catalog_sales* relations of TPC-DS 10GB database which are the largest relations of this database.

For each *R.attr*, we experimented with different a values corresponding to varying selectivities in the query and take the average of c_r values obtained for different a as the calibrated c_r for that *R.attr*. Table 5.5 shows calibrated values of *random_page_cost* for different *R.attr* and *correlation*. We can see that for different *R.attr* and correlation values, we obtain different values of c_r. Since we have isolated the impact of *locality of reference* from other two parameters, the only reason for obtaining different c_r values for different queries can be the difference in *locality of reference*.

Thus, we have experimentally verified that *locality of reference*, including *cache_hit_rate* is not correctly captured by PostgreSQL. If it was captured correctly, we would have obtained similar values of c_r in all the queries. The function we tried to model in Section 5.4.2 will help in correction in *locality of reference* at higher level, which is captured by *correlation* but *cache_hit_rate* still needs to be corrected and we propose an idea for it in the next subsection. The way we ensured correct *locality of reference* in $Cost2_{pages\_est}$ and $Cost2_{pages\_corr}$ of Section

Table 5.5: *Correlation* and *random_page_cost*

R.attr	***Correlation***	$\mathbf{c_r}$
ss_store_sk	0.018	1.495
cs_call_center_sk	0.082	0.442
cs_catalog_page_sk	0.980	0.381
ss_ticket_number	1.000	0.145

5.4.1 is discussed here. The queries used in this section for calibrating c_r are the ones used in Section 5.4.1 and since these different c_r values capture the different values of *locality of reference* for different queries, we could ensure correct *locality of reference* for $Cost2_{pages\_est}$ and $Cost2_{pages\_corr}$ when we substitute the calibrated c_r values obtained here, in them. This is also the reason we mentioned that $Cost2_{pages\_corr}$ shown in Table 5.3 is the best prediction possible because the values of c_r are substituted in the calibration queries itself, but it was shown just to indicate that proper prediction by cost model is possible if both *pages_fetched* and *locality of reference* are correct.

5.5.2 Using *correlation* to find *cache_hit_rate*

There is no guarantee that assumption of the optimizer designers that *cache_hit_rate* is 90% for all queries is correct. It should depend on the query in consideration. For example, in the queries in Section 5.1, *cache_hit_rates* were close to 0%. Thus, we need some measure to predict *cache_hit_rate* of the query to be executed.

PostgreSQL takes into account locality of reference at a higher granularity through the use

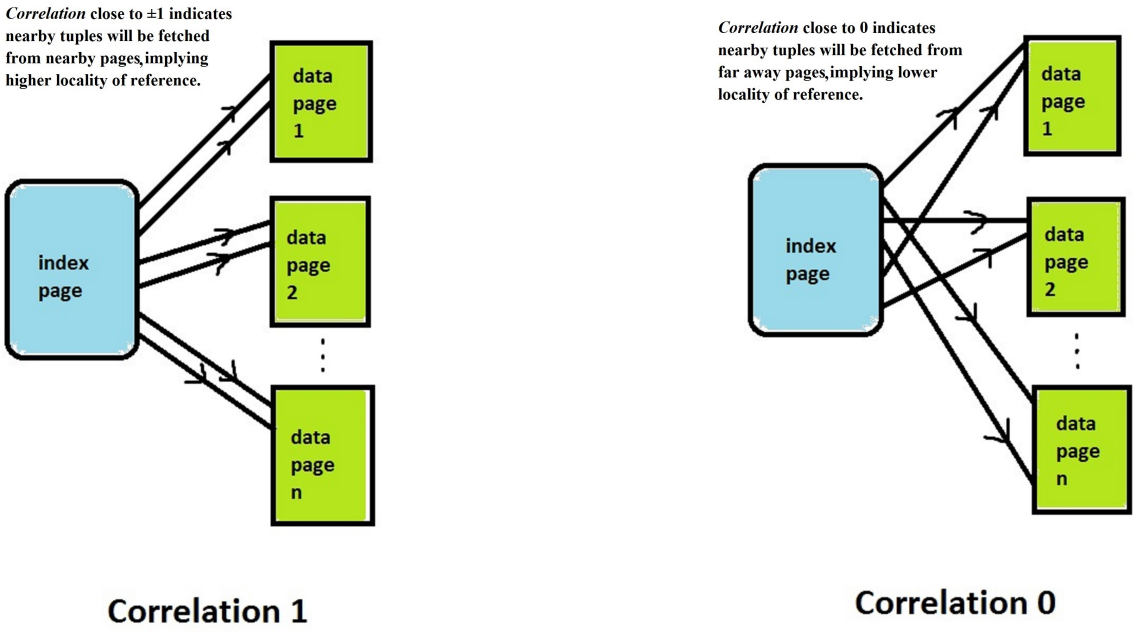


Figure 5.2: Relation between *correlation* and *cache_hit_rate*

of *correlation* as discussed in Section 5.2. *Correlation* accounts for sequential accesses for the cases when nearby tuple are fetched from nearby pages. But locality of reference at a lower granularity should also be taken into account, where nearby tuples are fetched from the pages already in the buffer cache. This locality of reference is captured by *cache_hit_rate*. Thus, both *correlation* and *cache_hit_rate* account for locality of reference and thus, are similar.

Also, as explained by Figure 5.2, if the value of *correlation* between physical row ordering and logical ordering of column values for an attribute is close to 0, nearby tuples will be fetched from pages far way from each other and there will be low locality of reference. If *correlation* is close to 1, nearby tuples will be fetched from pages which are physically close to each other and there will be high locality of reference. So *correlation* is a parameter which indicates locality of reference and hence, can be used as an indicator of *cache_hit_rate*.

Thus, both *correlation* and *cache_hit_rate* account for locality of reference and hence, are similar. PostgreSQL uses *correlation* for taking locality of reference into account at a higher granularity and at lower granularity, it assumes *cache_hit_rate* to be 90% always, which is not correct. So, locality of reference needs to be taken into account, depending upon the query, at lower granularity also. Since *correlation* and *cache_hit_rate* are similar, *correlation* is the parameter which we propose to use for taking locality of reference at this lower granularity, or

cache_hit_rate into account.

In Section 5.2, we discussed that cost for random page access is calculated as:

$$N3^{random} * C_R * 0.10 \tag{5.13}$$

We can rewrite this equation as:

$$C_R * N3^{random} * 0.10 \tag{5.14}$$

or,

$$C_R * (N3^{random} * 0.10) \tag{5.15}$$

Thus effectively, one-tenth of the pages are assumed to be randomly accessed. We denote this number of pages by $N4^{random}$ and they are defined by PostgreSQL as:

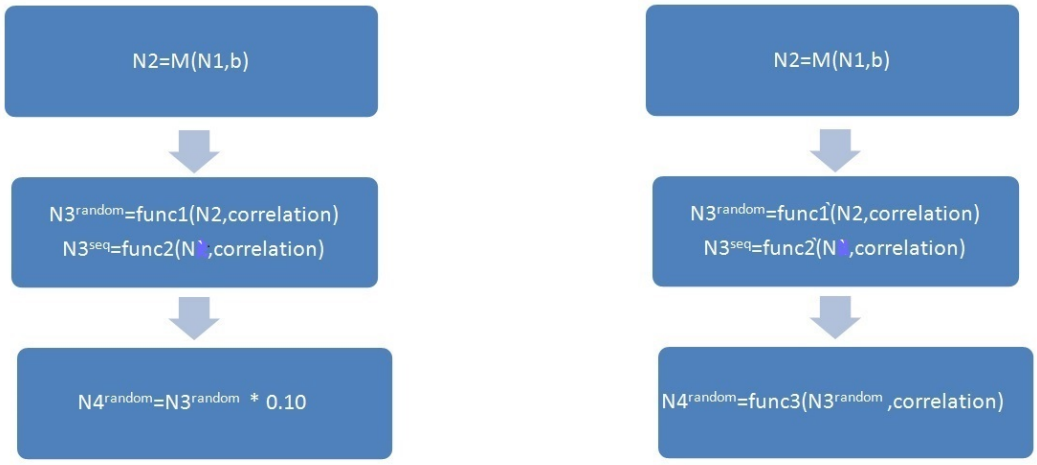
$$N4^{random} = N3^{random} * 0.10 \tag{5.16}$$

Here, 0.10 is the *cache_miss_rate* assumed by the optimizer designers. Our proposal is to use a function of *correlation* to estimate $N4^{random}$ instead of defining it as in Equation (5.16):

$$N4^{random} = func3(N3^{random}, correlation) \tag{5.17}$$

The *func3* stated above is a function of *correlation*, which should be defined in such a way as to indicate *cache_hit_rate* for the query. Modeling of *func3* can be done by measuring *cache_hit_rate* for some queries. This would require implementation of some counters at the O.S. cache levels and is not done by us.

The above discussion is summarized in Figure 5.3. Currently PostgreSQL uses *correlation* at one place to interpolate between costs for random and sequential access on execution of a query. Our proposal is to use *correlation* at one more place for taking *cache_hit_rate* of the query into account.



a) PostgreSQL implementation

b) Our proposal

Figure 5.3: Estimation of number of *pages_fetched* and *locality of reference* in PostgreSQL

Chapter 6

Calibrating Cost Model Using CPU Ticks

Since we are dealing with measurement of execution time, it is important to do the measurement accurately and precisely. According to [6], it is hard to do accurate and precise measurement of query execution time and they propose a technique for doing so.

It is tiresome to follow this approach for all the queries. But time measurements are crucial during calibration. So we followed their approach only for the calibration queries for getting a good calibration. We observed that the values of cost parameters obtained through this experiment are significantly different from the values obtained earlier. The reason for this is explained below.

We define *query_execution_time* as time taken for query execution only. This excludes time for triggers or acquiring/releasing of resources. *Gap_time* is defined as the time taken for triggers or acquiring/releasing of resources during query execution. *Background_processes_time* is defined as the time taken by the O.S. background processes. Lets denote the time obtained through the approach followed in [6] as *Metrology_time*. PostgreSQL gives us two types of time during query execution: (i) *actual time*, which is the same as *query_execution_time* and we call this as *Postgres_actual_time* (ii) *total time*, which is *query_execution_time* and *gap_time* together and we call it as *Postgres_total_time*. These two time intervals also include the time taken for O.S. background processes. Thus

$$Metrology\_time = query\_execution\_time + gap\_time \tag{6.1}$$

$$Postgres\_actual\_time = query\_execution\_time + background\_processes\_time \tag{6.2}$$

$$Postgres\_total\_time = query\_execution\_time + gap\_time + background\_processes\_time \quad (6.3)$$

For calibration, *query_execution_time* is the most appropriate time interval to use and while calibrating through the approach of [8] earlier, we were using *Postgres_actual_time* which is different from *Metrology_time* and hence we obtained different values for cost parameters through the two approaches.

However, precise and accurate time measurement for the calibration queries can be done if we follow the approach described in [6], but execution time is not taken as *Metrology_time*, rather

$$execution \quad time = Metrology\_time - (Postgres\_total\_time - Postgres\_actual\_time) \quad (6.4)$$

This gives us *query_execution_time*. This is explained by Figure 6.1. All the three readings: *Metrology_time*, *Postgres_total_time* and *Postgres_actual_time* have to be taken simultaneously for each query execution.

This whole procedure of calibration using ticks is time-consuming. We are working in a controlled environment, where we take multiple readings of execution time (*Postgres_actual_time*) for a query and drop the readings showing significant deviations from majority of the readings. Then average over the remaining executions is taken. Moreover, cost-modeling errors are much higher than the error contributed by the measurement of execution time for calibration queries in this environment. So the approach of calibration using ticks will not give much different results than the ones we are getting in this controlled environment.

However, in an uncontrolled environment where many O.S. background processes are running and readings of execution time for a query show significant variations, the approach using ticks which is discussed above can be used for calibration.

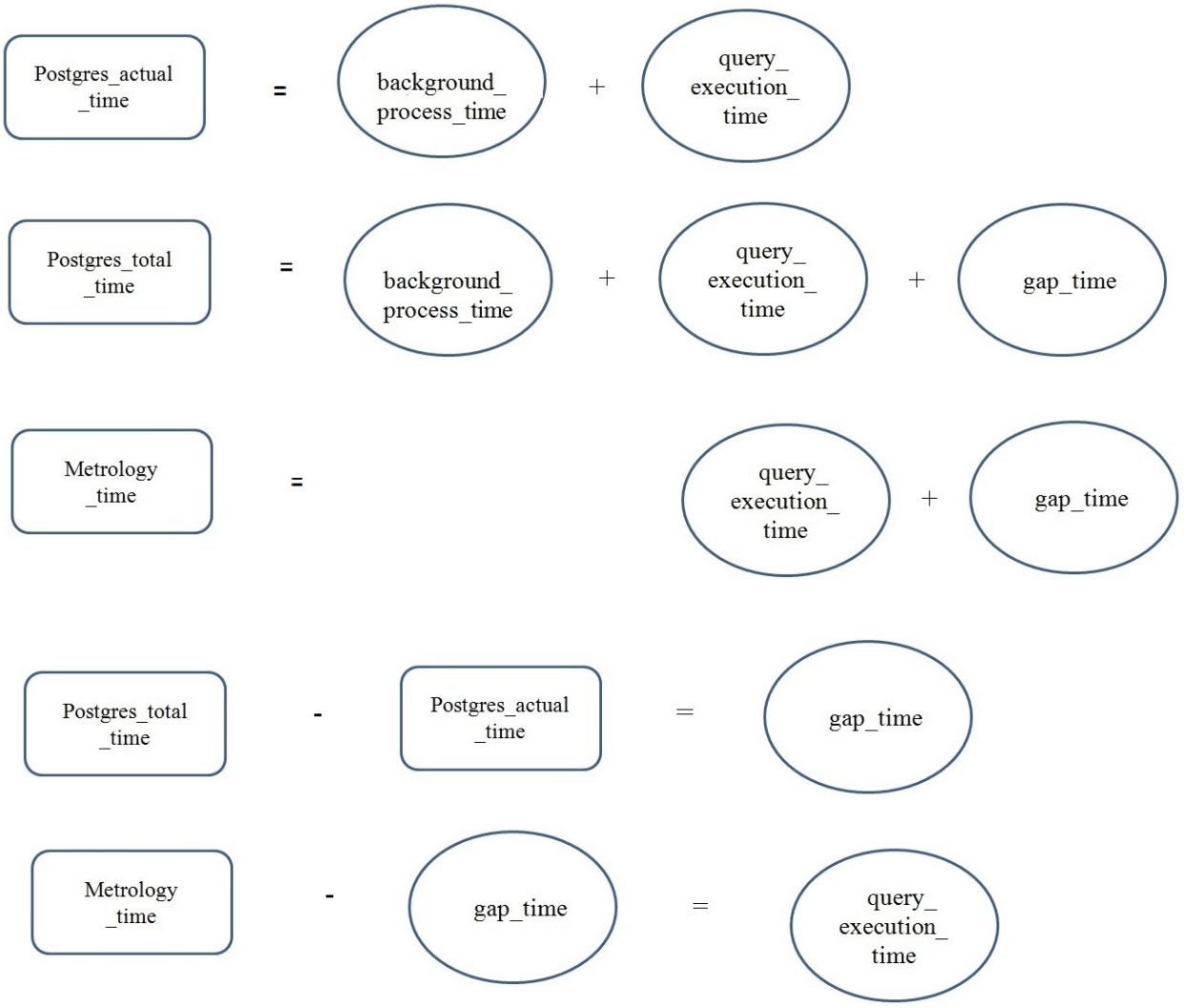


Figure 6.1: Measurement of execution time for calibration queries

Chapter 7

Experiments

In this chapter, we show our experimental setup and results obtained after calibrating the static parameters related to c_o.

7.1 Experimental Setup

PC: configured with a dual core 2.30 GHz Intel (Core i5) CPU and 4 GB memory
Linux Kernel: 3.2.0-23-generic-pae
PostgreSQL version: 9.0.4
Databases: TPC-H 1GB (Sections 7.2.1 - 7.2.3)
TPC-DS 10GB (Section 7.2.4)
Queries: 20 benchmark queries of TPC-H and
some benchmark queries of TPC-DS, not involving c_r.
The indexes specified by TPC-H/TPC-DS specification [4, 3] are created on the databases.

Each query is executed multiple times after clearing the file system and database buffers. While executing the queries, we tried to keep the system in a state such that no significant process was running on it except for the PostgreSQL (and some O.S. background processes which could not be killed). We did multiple executions of each query and the executions showing significant variation from execution times of majority of the executions were dropped. We took five execution time measurements from the remaining executions and their mean execution time is taken. This was a controlled environment. Correct and precise measurement of execution time is crucial in these experiments, so in an uncontrolled environment, where execution time of queries are showing major deviations, a more careful approach is required for this measurement, as described in Chapter 6 and based on the approach of [6].

Cost is obtained from the optimizer after explicitly substituting true cardinalities in the

query plan, obtained through actual pre-running of the queries.

7.2 Analysis of Static Parameters related to *cpu_operator_cost*

7.2.1 Calibration of Static Parameters

The values of calibrated tunable parameters were shown in Table 2.1 in Section 2.3.1. The values of calibrated static parameters of the cost model, obtained on our system, are shown in this section in Table 7.1. Column 2 shows the actual values of static parameters obtained for the three types of operations and Column 3 shows the ratios among these values normalized with respect to *seq_page_cost*.

The tunable parameters of $Calib_{static}$ *model* have the same values as for $Calib_{tunable}$ *model* and shown in Table 2.1.

Table 7.1: Values of static parameters for Different Operators

Operators	**Values of Static Parameters**	**Ratios among the Parameters**
Sort	3.14e-5	2.99e-4
Nested Loop Join	1.41e-4	1.34e-3
Generic	2.13e-4	2.03e-3

7.2.2 Evaluation on TPC-H Query Template

In this section, we show experimental results with a TPC-H query template. We define *Error Diagram* to be a three-dimensional visualization of relative errors over the relational selectivity space. In this, one axis shows the relative error associated with prediction of query execution time at different query points. The other two axes show the varying relational selectivities over some selected predicates.

Figure 7.1 shows the *Error Diagram* corresponding to QT3 based on Query 3 of TPCH-benchmark, where x and y axes show varying selectivities over ORDERS and LINEITEM relations over o_totalprice:varies and l_extendedprice:varies predicates respectively. Relative prediction error for each query is plotted on the z-axis. In both the cost models, cost is always overestimating time for these queries.

From the *Error Diagram* in Figure 7.1, we can see that the relative error obtained using $Calib_{tunable}$ *model* increases with increasing relational selectivities. But the error using

$Calib_{static}$ *model* does not increase much and is always below the error using $Calib_{tunable}$ *model*.

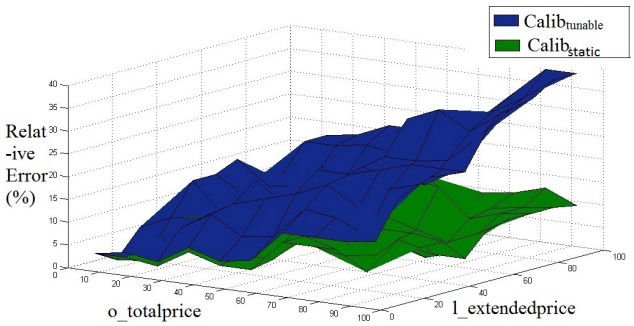


Figure 7.1: Error Diagram for TPC-H Query Template 3

The reason behind it is that a *sort* operator is used in these queries near the root of the plan tree. The $Calib_{tunable}$ *model* does not include calibration of static parameter for sort and hence, introduces some error in predicting query execution time. As the relational selectivities increase, more number of tuples are returned to the root, thereby increasing contribution of sort and hence, the error. But introduction of m_o^{sort} in $Calib_{static}$ *model* handles the error due to sort and hence, gives lower error even at higher selectivities. Although overall error with $Calib_{tunable}$ *model* remains around 40% at highest relational selectivities , error imparted by sort increases upto 200%, whereas this error remains only 22% with the $Calib_{static}$ *model*.

7.2.3 Evaluation on TPC-H Benchmark Queries

In this section, we show the relative errors in prediction of execution time by the cost model on TPC-H benchmark queries with $Calib_{static}$ *model*. With $Calib_{static}$ *model*, in cases where $m_o^{generic}$ is being significantly used by the optimizer in cost estimation, performance is comparable to $Calib_{tunable}$ *model*. So we do not show the results on queries where error was same as with $Calib_{tunable}$ *model* as these are the same values as shown in Table 2.2 of Section 2.3.2.

Table 7.2 shows these cost-modeling errors. Column 2 shows the errors with $Calib_{tunable}$ *model* obtained using the approach of [8] on our system, while Column 3 shows the errors with $Calib_{static}$ *model*. Err_{calib_tun} and Err_{calib_static} denote the relative error with $Calib_{tunable}$ *model* and the relative error with $Calib_{static}$ *model* respectively. With both the cost models, in most of the queries, cost is overestimating time, while for few queries it is underestimating.

$Calib_{tunable}$ *model* shows significant improvement over default parameters on average which is evident from the much lower MRE for it. $Calib_{static}$ *model* performs as good as the $Calib_{tunable}$ *model* in most cases (all cases, except query 16) and shows significant improvement in some cases.

Table 7.2: *Relative Errors* for TPC-H Benchmark Queries

Query	$\boldsymbol{Err_{\text{calib\_tun}}}$	$\boldsymbol{Err_{\text{calib\_static}}}$
10	0.12	0.08
11	0.57	0.51
13	1.84	0.42
16	-0.35	-0.44
MRE	**0.40**	**0.32**

MRE is reduced from 0.81 in case of the default cost model to 0.40 by using $Calib_{tunable}$ *model* of [8]. This was further reduced by our $Calib_{static}$ *model* to 0.32. Thus, our cost model is at least as good as $Calib_{tunable}$ *model* and significantly better in some cases.

7.2.4 Evaluation on TPC-DS Benchmark Queries

Earlier we were dealing with TPC-H 1GB database which could entirely fit into memory of the system. We decided to switch to a larger and more skewed database for testing effectiveness of the cost model.

We tested the prediction of execution time by the cost model on some benchmark queries of TPC-DS. In Table 7.3, we show the results on some benchmark queries not involving c_r. This was done to check if the calibrated cost models work well on queries over this database in case no random access is there. As with TPC-H 1GB, these queries performed well, giving small relative errors, except for query 62.

The reason for high prediction error for query 62 by $Calib_{tunable}$ *model* is that we forced sequential scan and merge join on this query, making the contribution of sort significant. So $Calib_{tunable}$ *model* gives a huge prediction error, while $Calib_{static}$ *model* performs good as it has calibrated value of m_o^{Sort}.

Table 7.3: *Relative Errors* for TPC-DS 10GB Benchmark queries

Query	$\boldsymbol{Err_{\text{calib\_tun}}}$	$\boldsymbol{Err_{\text{calib\_static}}}$
7	0.07	0.07
21	0.35	0.34
28	0.40	0.40
43	0.09	0.09
62	**1.75**	**0.04**
66	0.03	0.03
96	0.03	0.03
98	-0.09	-0.09
MRE	**0.35**	**0.14**

Chapter 8

Conclusions

We are working in the direction of finding a good PostgreSQL cost model and we started with the calibration idea of [8]. In [8], the authors propose a $Calib_{tunable}$ *model* where they find the appropriate values of tunable parameters taking the hardware configurations of the system on which queries have to be executed into account. Along with these tunable parameters, we also consider the static parameters and propose a $Calib_{static}$ *model*. Through our experiments, we show that it provides at least as good execution time prediction as the $Calib_{tunable}$ *model* in most of the cases and significantly better than it in some cases. This cost model is better than the $Calib_{tunable}$ *model* suggested in [8] as can be seen from the values of MRE in Tables 7.2 and 7.3, and the results in Section 7.2.2 for QT3.

We designed a query which could achieve pure random access and hence, calibrated *random_page_cost*. We experimentally explored the reasons for poor prediction by this parameter and found that three parameters associated with *random_page_cost* can cause prediction errors. We further found that out of the three factors responsible for error in cost due to c_r: (i) *random_disk_access_time*, (ii) *pages_fetched*, and (iii) *locality of reference*, *random_disk_access_time* is correct. *Pages_fetched* and *locality of reference* need correction. We tried to model some functions of *correlation* in order to get a good estimate of *pages_fetched*. We propose to use *correlation* to predict the *cache_hit_rate* of queries.

Bibliography

[1] http://www.postgresql.org/docs/9.0/static/runtime-config-query.html.

[2] http://en.wikipedia.org/wiki/Pigeonhole_principle.

[3] http://www.tpc.org/tpcds/, .

[4] http://www.tpc.org/tpch/, .

[5] http://www.postgresql.org/docs/9.0/static/view-pg-stats.html.

[6] S. Currim, R. T. Snodgrass, Y-K. Suh, R. Zhang, M. W. Johnson, and C. Yi. Dbms metrology: Measuring query time. *SIGMOD*, 2013.

[7] A. Dutt and J.R. Haritsa. Plan bouquets: Query processing without selectivity estimation. *SIGMOD*, 2014.

[8] W. Wu, S. Zhu Y. Chi, J. Tatemura, H. Hacigümüş, and J.F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? *ICDE*, 2013.