# Think Global, Model Local: A Fine-Grained Approach to Query Cost Estimation with Learned Parameters

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

**Master of Technology**

IN

**Faculty of Engineering**

BY

**Vishal Goel**

Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

July, 2020

# Declaration of Originality

I, **Vishal Goel**, with SR No. **04-04-00-10-42-18-1-15451** hereby declare that the material presented in the thesis titled

**Think Global, Model Local: A Fine-Grained Approach to Query Cost Estimation with Learned Parameters**

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-2020**.
With my signature, I certify that:

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

Date: Student Signature

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

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

*my beloved family and friends*

# Acknowledgements

I am immensely grateful to my advisor, Prof. Jayant R. Haritsa, for providing me with an opportunity to work under his esteemed guidance. Among the many teachings he has imparted over the years, approaching a problem in the simplest way possible and presenting your solution with the utmost clarity of thought and elegance are the golden ideas I will never forget.

I also thank the Database Systems Lab for providing me with an unwavering support and motivation to keep moving forward, one step after the other.

I further thank the Department of Computer Science and Automation and Indian Institute of Science for an excellent platform conducive to a steady learning growth and a peaceful atmosphere for true mindfulness.

Finally, I pay my deepest respects to my family and friends for always believing in me.

# Abstract

Prediction of query execution time has been a problem of vital importance since the very time database management systems came into existence. Other than the obvious benefit of knowing when the query would finish executing, it helps in query optimisation and taking decisions concerning admission control, query scheduling and system sizing. The problem of predicting query execution time is challenging particularly because it involves predicting runtime uncertainties like data access patterns, data skew, spilling of data from memory to disk, etc. The literature on this problem can be categorised into two primary approaches – (a) tuning the existing cost models and (a) machine learning (ML) based models. The former approach tunes the cost parameters used in cost functions of the existing model and the ML models are trained on a set of executed query plans to predict execution time of a new plan. Thus, while the white box approach of tuning achieves a very limited accuracy without questioning the fundamental cost functions in the cost model, the ML models (which treat DBMS as a black box) are highly training hungry, lack explainability and risk performing inadequately on queries dissimilar to the training queries. In this work, after carefully investigating a well-tuned PostgreSQL engine's cost model and trying to fix it with minimal changes, we found that for most operators, replacing PostgreSQL's analytical cost functions with more sophisticated functions (and replacing global cost parameters with new learnable local cost parameters specific to each operator) can predict the execution time accurately. And for one particular operator (index scan), a simple explainable learning approach like piecewise linear approximation is necessary and sufficient. Our approach to cost modelling is a white-box approach that builds on top of years of expert knowledge, is explainable (i.e. can tell exactly where and why each second was spent during query execution), generalisable to any kind of new queries and easily integrable into traditional database systems. We evaluated our model against a well-tuned PostgreSQL's cost model on the realistic Join Order Benchmark and have been able to predict query execution time very close to accurate (bringing PostgreSQL's mean Qerror of 7.35 down to 1.3) on a set of more than 350 query plans.

# Contents

**Acknowledgements** **i**

**Abstract** **ii**

**Contents** **iii**

**List of Figures** **v**

**1 Introduction** **1**

1.1 Characteristics of Our Approach . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Roadmap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 PostgreSQL's Cost Model** **5**

2.1 Tuning PostgreSQL's Cost Parameters . . . . . . . . . . . . . . . . . . . . . 5

2.2 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Prediction of Scan Operators** **8**

3.1 Sequential Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.2 Index Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.2.1 PostgreSQL's Cost Model of Index Scan . . . . . . . . . . . . . . . . . . 9

3.2.1.1 Shortcomings . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.2.2 Our Cost Model of Index Scan . . . . . . . . . . . . . . . . . . . . . . . 11

3.2.2.1 Training Procedure . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.2.2.2 Prediction Procedure . . . . . . . . . . . . . . . . . . . . . . . . 14

**4 Prediction of Sort Operator** **15**

4.1 PostgreSQL's Cost Model for Sort . . . . . . . . . . . . . . . . . . . . . . . . 15

4.1.1 Shortcomings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.2 Our Cost Model for Sort . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**5 Prediction of Join Operators** **20**
5.1 Nested Loop Join . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.1.1 PostgreSQL's Cost Model for Nested Loop Join . . . . . . . . . . . . . . . . 20
5.1.2 Our Cost Model for Nested Loop Join . . . . . . . . . . . . . . . . . . . . 21
5.2 Sort Merge Join . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.2.1 PostgreSQL's Cost Model for Sort Merge Join . . . . . . . . . . . . . . . . 23
5.2.2 Our Cost Model for Sort Merge Join . . . . . . . . . . . . . . . . . . . . . 23

**6 Prediction of Group By Operator with Aggregation** **25**

**7 Experiments** **27**
7.1 Evaluation Benchmark . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
7.2 Performance Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

**8 Spilling** **33**

**9 Related Work** **36**

**10 Conclusion and Future Work** **37**

**Bibliography** **39**

# List of Figures

2.1 Predictions of query execution times on two example plans . . . . . . . . . . . . 6

3.1 Scan Query Template $Q\langle R, A, v1, v2\rangle$ . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Analysis of PostgreSQL's cost model for Index Scan . . . . . . . . . . . . . . . . 9
3.3 Training Curves 1 and 2 for Prediction of Range Index Scan . . . . . . . . . . . 12
3.4 Training Curves 3 and 4 for Prediction of Range Index Scan . . . . . . . . . . . 13

4.1 Sort Query Template $Q\langle R, \{A1, A2, .., Ak\}\rangle$ . . . . . . . . . . . . . . . . . . . . 15
4.2 Sort Operator Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.3 Heuristics for change in corr and dup values of an attribute in a plan . . . . . . 18

5.1 Join Query Template $Q\langle R1, R2, \{\{A1, B1\}, .., \{Ak, Bk\}\}\rangle$ . . . . . . . . . . . . . 20
5.2 Nested Loop Join Operator Examples . . . . . . . . . . . . . . . . . . . . . . . 22
5.3 Sort Merge Join Operator Examples . . . . . . . . . . . . . . . . . . . . . . . . 24

6.1 Group By with Aggregate Query Template $Q\langle R, \{A1, A2, .., Ak\}, \{B1, B2, .., Bm\}\rangle$ 25

7.1 Performance on Suite 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
7.2 Performance on Suite 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
7.3 Performance on Suite 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
7.4 Performance on Suite 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
7.5 Performance on Suite 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
7.6 Performance on Suite 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

8.1 External Sort Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
8.2 Spilling query examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

10.1 Summary of our approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

# Chapter 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:** Among various possible plans to execute a query, the optimiser picks the one with the least estimated cost.

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

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

However, runtime uncertainties (like how the data will be accessed and where it will be accessed from) and data properties (like skewness and duplicates) add to the non-deterministic nature of the problem. The literature on this problem can be broadly categorised into two primary approaches – (a) tuning the existing cost models ([15], [16]) and (b) machine learning (ML) based models ([1], [3], [4], [7], [9], [10], [12], [13], [14]). While the former approach tunes the cost parameters used in cost functions of the existing model, the ML models are trained on a set of executed query plans to predict execution time of a new plan. Thus, the white box approach of tuning achieves a very limited accuracy because it does not question the fundamental cost functions in the cost model itself, and the ML approach (which treats the DBMS as a black box) is highly training hungry, lacks explainability and performs poorly ([13], [14], [15]) on queries dissimilar to the training queries.

In this work, we carefully investigate where a well-tuned PostgreSQL engine falls short, if at all, in predicting the execution time of a given query and try to fix it with the least possible number of changes. Since PostgreSQL follows an operator-level cost modelling approach, we first look for what all really needs prediction in the first place to predict execution time of each operator, then present cost model of each operator (Sequential Scan, Index Scan, Sort, Nested Loop Join, Sort Merge Join and Group By with Aggregation), followed by how cost models of individual operators can be combined to predict execution time of the whole plan.

## 1.1 Characteristics of Our Approach

Following are the characteristics of our approach towards prediction of query execution time:

**Leveraging system knowledge:** Instead of building an altogether new cost model (like the ML models that treat DBMSes as black box), we leverage years of expert knowledge put into cost modelling and build on top of it (in other words, by taking the cost model of PostgreSQL as the starting point). This approach has shown that replacing analytical cost functions of most of the operators with more sophisticated ones and replacing cost model of one operator (index scan) with a simple ML model is sufficient for fairly accurate predictions. This approach further makes it easy to integrate into the existing engine's cost model.

**Changing cost functions:** We change the cost functions PostgreSQL uses for each operator and learn the cost parameters used in these functions either by running calibrating queries or using ML models. For example, for index scan, we replace the analytical model of PostgreSQL based on unrealistic assumptions (like uniform tuple distribution across data pages and uniform duplicate distribution in attribute domain) with an ML model that learns execution time of index scan by training on domain-covering index scans that learn properties of the physical layout of the database. For sort operator, we add new cost functions based on pre-sortedness and duplicates in the attribute values. And for join operators, we tune the cost parameters locally using calibrating queries. Thus, our approach is fundamentally different from the existing approaches that either treat the system as a black box or tune the cost parameters globally leaving the cost functions unchanged.

**Explainability:** For most of the operators, our analytical models predict execution times accurately. For index scan, learning on the physical layout of data on disk is imperative, as we will show later, which we achieve by using piecewise linear approximation on a training set of domain-covering index scans. This hybrid approach of sticking to analytical modelling and using explainable ML modelling sparingly makes it easy to put finger on where exactly each second of the query execution time would be spent and, in case the prediction goes awry, where exactly it went wrong, as we will show in Chapter 7. Furthermore, if one still wants to exploit deep learning, there is no need to train on whole plans but only on index scan operator. This way, the accuracy, explainablity and generalizability of the deep learning models themselves can be enhanced.

**Generalizability:** The ML models risk performing not as good ([13], [14], [15]) on plans dissimilar to the ones used in training set. However, our approach is completely generalizable because we do not train on a set of executed plans. Only for index scan, we train on a set of index scan queries that cover whole domain of attributes. Note that it is far difficult to cover the plan space using a set of executed plans than cover an attribute domain space using a set of range index scans. In Chapter 7, we tested our model on unseen query plans and predicted accurately for most of them.

## 1.2 Assumptions

We have taken the following assumptions for our query execution environment: a) Single query and isolated environment (i.e. no other user processes running on the system), b) Disk-resident database, c) Cold cache (i.e. memory is cleared before running a query), d) Indexes built only on numerical attributes and d) Perfect cardinality estimates. Each cost function takes the number of tuples (cardinality) to be processed as an input parameter. To separate errors of cost estimation from those of cardinality estimation, we assume that the input cardinalities are perfect. To actualise this assumption in practice, we first force the executor to execute a given query plan, then record the actual cardinalities at each edge, then feed these perfect cardinialities back into the cost model (bypassing the calls to the cardinality estimator) thus giving us predicted costs at each edge of the query plan on perfect cardinality inputs. Further, we only consider the operators - Sequential Scan, Index Scan, Sort, Nested Loop Join, Merge Join and Group By with Aggregation - in this work. Although single query environment might seem like an unrealistic assumption from industrial perspective, we want to first test our approach in this simpler environment and then move forward to multi-query (resource-sharing) environment.

## 1.3 Roadmap

We first briefly explain how PostgreSQL estimates the cost of a query plan and how Wu's model [15] tunes the parameters in Chapter 2. Then we talk about prediction of each operator's execution time (Sequential and Index Scan in Chapter 3, Sort in Chapter 4, Nested Loop Join and Sort Merge Join in Chapter 5, and Group By with Aggregation in Chapter 6). For each operator, we talk about PostgreSQL's execution and cost model, followed by its caveats and our modelling approach explained with a few examples on the IMDB [6] dataset. We assume a no spilling scenario (i.e. the memory does not have to spill data over to the disk during query execution due to lack of space) for chapters 3 to 6 and show experiments for the same in Chapter 7 where we evaluate our model against a well-tuned PostgreSQL's on the Join Order Benchmark (JOB [6]). Then, we discuss the spilling scenario in Chapter 8 and the related work in Chapter 9 followed by a note of conclusion and future work in Chapter 10.

# Chapter 2

# PostgreSQL's Cost Model

We start with giving a broad overview of PostgreSQL's cost model that uses a vector of five global cost parameters. Let $C$ be that vector, as follows:

$$C = [IO\_SEQ\_COST,\ IO\_RANDOM\_COST, CPU\_TUP\_COST,\ CPU\_INDEX\_COST,\ CPU\_OP\_COST]^T$$

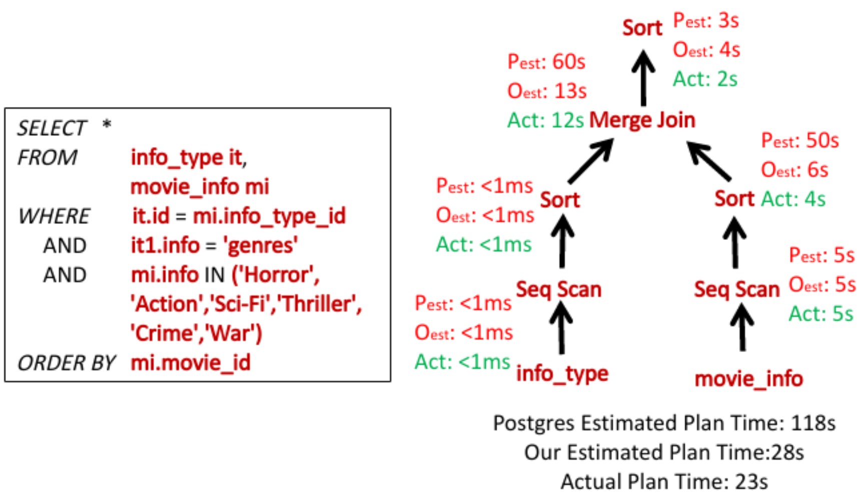
where the cost parameters are defined as:

1. $IO\_SEQ\_COST$: I/O cost to sequentially access a page
2. $IO\_RANDOM\_COST$: I/O cost to randomly access a page
3. $CPU\_TUP\_COST$: CPU cost to process a relation tuple
4. $CPU\_INDEX\_COST$: CPU cost to process an index tuple
5. $CPU\_OP\_COST$: 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 $N = [N_s, N_r, N_t, N_i, N_o]^T$ 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 $O$.

## 2.1 Tuning PostgreSQL's Cost Parameters

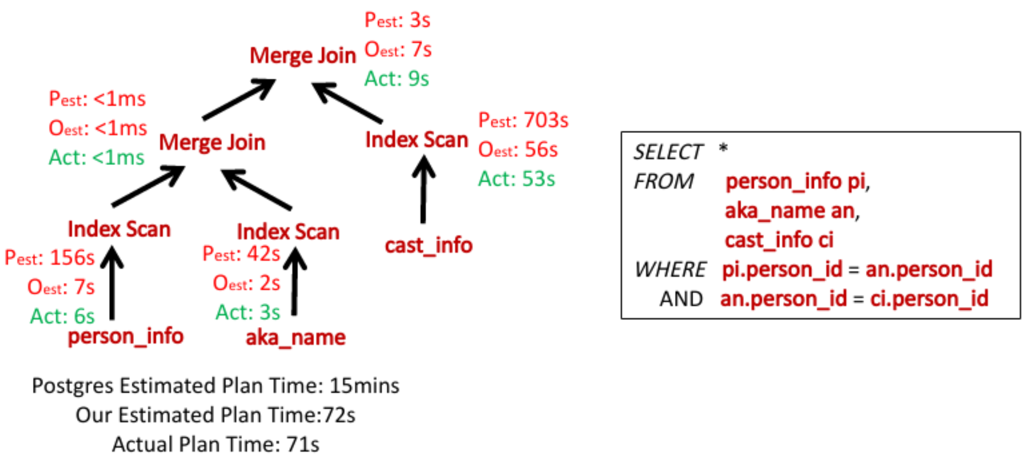
Postgres assigns default values to the vector $C$ ($[1,\ 4,\ 0.01,\ 0.05,\ 0.0025]^T$) keeping $IO\_SEQ\_COST$ as the base value and other parameters' values relative to it. Thus, the estimated cost of Postgres is in $IO\_SEQ\_COST$ units. To avoid assigning default values and to estimate cost in real time units, Wu et al [15] tune these parameters using a set of calibrating queries. To get the cost of the whole plan, Postgres adds the cost of the individual operators. This is because Postgres uses a single-threaded process to execute a query plan. Thus, at one time, only one operator operates resulting in no parallel computations. Further, since it is a pull-based execution model (i.e. to get a tuple, the parent node has to ask the child node), CPU computations and disk fetches do not overlap.



```
SELECT    *
FROM      info_type it,
          movie_info mi
WHERE     it.id = mi.info_type_id
  AND     it1.info = 'genres'
  AND     mi.info IN ('Horror',
          'Action','Sci-Fi','Thriller',
          'Crime','War')
ORDER BY  mi.movie_id
```

Sort Pest: 3s Oest: 4s Act: 2s
Merge Join Pest: 60s Oest: 13s Act: 12s
Sort Pest: <1ms Oest: <1ms Act: <1ms
Sort Pest: 50s Oest: 6s Act: 4s
Seq Scan Pest: <1ms Oest: <1ms Act: <1ms
Seq Scan Pest: 5s Oest: 5s Act: 5s
info_type
movie_info
Postgres Estimated Plan Time: 118s
Our Estimated Plan Time:28s
Actual Plan Time: 23s

(a) Query Cost Prediction Example 1



```
SELECT    *
FROM      person_info pi,
          aka_name an,
          cast_info ci
WHERE     pi.person_id = an.person_id
  AND     an.person_id = ci.person_id
```

Merge Join Pest: 3s Oest: 7s Act: 9s
Merge Join Pest: <1ms Oest: <1ms Act: <1ms
Index Scan Pest: 703s Oest: 56s Act: 53s
Index Scan Pest: 156s Oest: 7s Act: 6s
Index Scan Pest: 42s Oest: 2s Act: 3s
person_info
aka_name
cast_info
Postgres Estimated Plan Time: 15mins
Our Estimated Plan Time:72s
Actual Plan Time: 71s

(b) Query Cost Prediction Example 2

Figure 2.1: Predictions of query execution times on two example plans

## 2.2 Examples

Figure 2.1 shows an example of two queries (part of JOB queries 18b and 7a) executed on Postgres. Figures 2.1a and 2.1b show their query plans annotated with the Wu-tuned PostgreSQL's estimations ($P_{est}$) and our estimations ($O_{est}$) for each operator, along with the actual execution times ($Act$). In Example 1, Postgres overestimated the cost of query plan ( 2 mins instead of 23s), mostly due to overestimating the cost of all sort operators. This is broadly because Postgres charged a sort comparison cost ($CPU\_OP\_COST$) of eleven times higher than the actual sort comparison cost (which we locally tuned using a calibrating query to get the cost right). Similarly in Example 2, Postgres predicted the cost of plan as 15 mins whereas it was actually only 71s. This was mostly because it overestimated the costs of all Index Scan operators based on uniform tuple distribution assumption across all pages which led to overestimations in number of pages predicted to be scanned from the disk. The input cardinality estimates in each operator are ensured to be perfect, thus the errors are only due to cost modelling.

# Chapter 3

# Prediction of Scan Operators

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

Figure 3.1: Scan Query Template $Q\langle R, A, v1, v2\rangle$

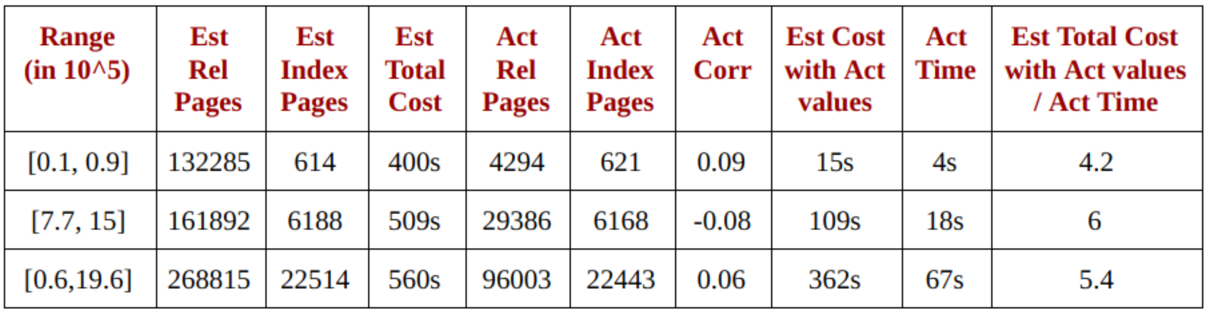
Scan operators can take up a lot of total execution time of a query, especially in the case of disk-resident databases, and thus their accurate prediction is important. There are mainly two types of scan techniques that DBMSes use – sequential scan and index scan. For the sake of convenience, let us assume a SQL query $Q < R, A, v1, v2 >$ (Figure 3.1). Executing Q returns the tuples from relation R whose values of attribute A lie between $v1$ and $v2$ (both inclusive). Note that other forms of predicates (i.e. with $<, >, ==, != $) can be easily converted into the form of Q. Also, an absence of the upper (and/or lower bound) on the attribute value simply means presence of the default maximum (and/or minimum value) of the attribute. We will now discuss both the scan operators in detail.

## 3.1 Sequential Scan

Given a query $Q < R, A, v1, v2 >$, the sequential scan operator sequentially reads the consecutive pages of the relation R while filtering out tuples based on the given filter. Thus:

$$Estimated\ Cost = IO\_SEQ\_COST\ *\ n\ +\ CPU\_TUP\_COST\ *\ t$$

where $n$ = total pages and $t$ = total tuples in the relation R. We evaluated Wu-tuned PostgreSQL's predictions on all relations in JOB for sequential scan and all the queries were found to have a relative error of $\leq 1\%$. Thus, we use PostgreSQL's model for sequential scan as it is.



| Range (in 10^5) | Est Rel Pages | Est Index Pages | Est Total Cost | Act Rel Pages | Act Index Pages | Act Corr | Est Cost with Act values | Act Time | Est Total Cost with Act values / Act Time |
|---|---|---|---|---|---|---|---|---|---|
| [0.1, 0.9] | 132285 | 614 | 400s | 4294 | 621 | 0.09 | 15s | 4s | 4.2 |
| [7.7, 15] | 161892 | 6188 | 509s | 29386 | 6168 | -0.08 | 109s | 18s | 6 |
| [0.6,19.6] | 268815 | 22514 | 560s | 96003 | 22443 | 0.06 | 362s | 67s | 5.4 |

Figure 3.2: Analysis of PostgreSQL's cost model for Index Scan

## 3.2 Index Scan

Apart from the range filter predicate ($v1 \leq A \leq v2$), there can be another type of filter predicate: equality predicate ($A = v$). In isolation, the equality predicate can be seen as a range predicate ($v \leq A \leq v$), but in a plan setting, the equality predicate has to be treated differently. We defer this discussion to Section 5.1 and focus only on the range predicate here.

During range index scan, Postgres first reads the index pages 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 index leaf page and repeats the procedure until the the last value $v2$ has been read. The index scan operator is particularly hard to predict because Postgres accesses the data pages in ascending order of attribute values from $v1$ to $v2$, physical locations of which can be in any order or even in different parts of the disk due to fragmentation. Due to this, time to access data pages can vary depending on the physical distance between them. Data pages once accessed previously may also be found cached in memory. Due to this, predicting the number of pages to be scanned from disk solely based on tuple cardinality is challenging.

### 3.2.1 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 scanning index pages from root to leaf, and the CPU cost of processing the index and relation tuples, which are negligible compared to I/O time of relation and index pages.

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

$$C(R) = corr^2(MinC) + MaxC(1 - corr^2)$$

$$MinC = relation\_selectivity\ * total\_relation\_pages\ *\ IO\_SEQ\_COST$$

$$MaxC = estimated\_relation\_pages\ * IO\_RANDOM\_COST$$

where $estimated\_relation\_pages$ is found by the Mackert and Lohman (M&L) approximation model [8]. Further, estimated cost to read index pages ($C(I)$) is given as:

$$C(I) = relation\_selectivity\ * total\_index\_pages\ *\ IO\_RANDOM\_COST$$

Thus, the total estimated cost by Postgres is given as:

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

#### 3.2.1.1 Shortcomings

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

1. It uses the same value of corr and $IO\_RANDOM\_COST$ for all ranges, which can be different for different ranges. For instance, in Figure 3.2, we ran a few index scan queries on the $movie\_id$ attribute of the $movie\_info$ relation (from the IMDB dataset) for which the estimated and actual values of the variables used in aforementioned cost model of index scan are shown. The corr value of the attribute ($movie\_id$) is 0.1 but the actual corr of the last query is almost half. Similarly, another range (in Figure 3.3a, Range:$[0, 250392]$, of which we will talk about later) has corr value of 0.003.

2. The M&L formula assumes a) uniform distribution of tuples across data pages, b) uniform duplicate distribution across attribute domain and that c) the memory buffer follows the LRU policy, among other assumptions, which are not very realistic. As seen in Figure 3.2, the estimated number of relation pages using this approximation model are far more than the actual number of relation pages scanned.

3. The cost model for index scan itself is questionable as it gives inaccurate estimations even after replacing the estimated values of all the above input parameters with actual values. Note that the ratio of the estimated total cost with actual values and actual time is also

different for the queries. Thus, a magic number to multiply with the estimated cost to get actual time is not possible.

### 3.2.2 Our Cost Model of Index Scan

PostgreSQL's model went for a toss because of its global view of the relation being scanned. To capture local view, we try partitioning the attribute domain such that in each partition, data follows similar properties like correlation, tuple distribution across pages and disk location. So, if a range falls inside a partition, it can be easily predicted (as we will show below) and if it falls across partitions, then we can predict contribution of each partition to the total execution time. Let us explain our training and prediction procedures using an example:

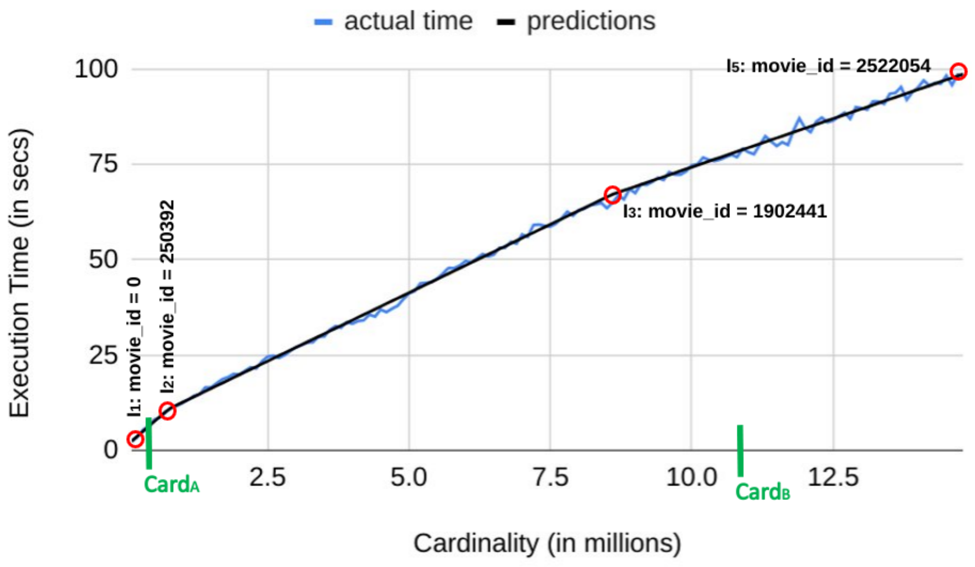
#### 3.2.2.1 Training Procedure

1. Generate the (blue-coloured) curve in Figure 3.3a by executing the following query:

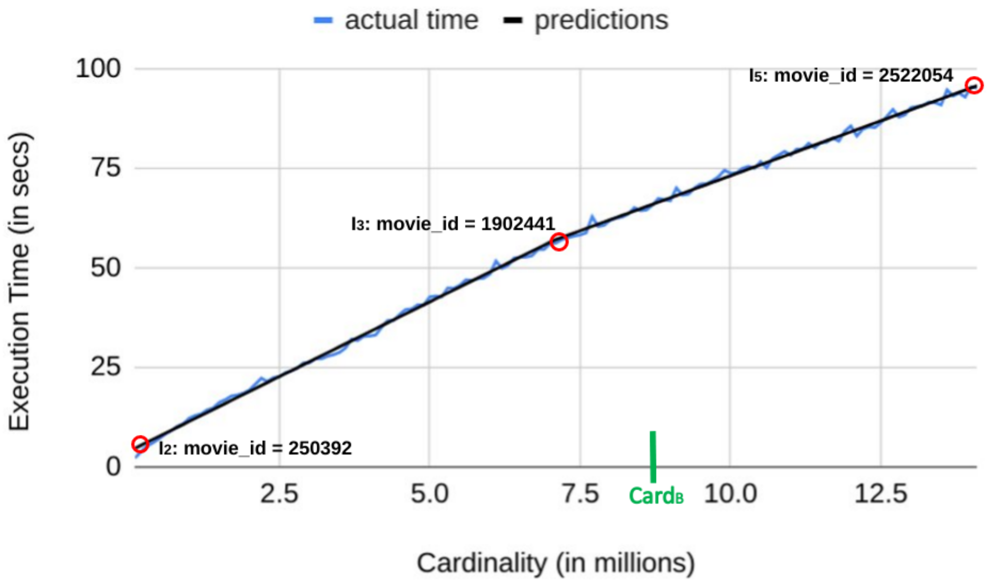
   $$select\ *\ from\ movie\_info\ where\ lb \leq movie\_id \leq v$$

   where $lb (= 0)$ is the lower bound on *movie_id* and variable $v$ is increased in a fixed intervals. Note that the x-axis is cardinality and not the attribute value. The resulting blue curve can be approximated with a linear piecewise function (shown in black colour). Here, three lines were sufficient giving us two inflexion points. For the sake of simplicity, let's call the starting point and end point as inflexion points as well. Thus, we have a total of four inflexion points - $I_1$, $I_2$, $I_3$ and $I_4$. The point $I_2$ here represents a change in the correlation. For range $I_1$ to $I_2$, the corr value was 0.003 but for range $I_2$ to $I_3$, the value increased to 0.06 due to proximity of data pages, thus giving a decrease in slope as the cardinality increased. Further, around 10% of the data pages scanned in range [$I_2$, $I_3$] were found cached in memory (already scanned in range [$I_1$, $I_2$]).

2. Suppose we want to predict execution time for a range between $I_1$ and $I_2$. Then, we do it only using the Curve 1. But if we want to predict for a range between, say $I_2$ and $I_3$, we cannot use only the Curve 1 as this will be an underestimation because of the decrease in slope due to caching effect. Thus, we need a fresh curve which starts from the point $I_2$. So, create a new curve (Figure 3.3b) with the second inflexion point $I_2$ (in Figure 3.3a) as the starting point. We obtained three inflexion points in Figure 3.3b after piecewise linear curve fitting. The corr value of range $I_2$ to $I_3$ was almost same as that of $I_3$ to $I_4$ but "half" of the data pages accessed in the latter range were already accessed in the former range, thus decreasing the slope due to caching effect.



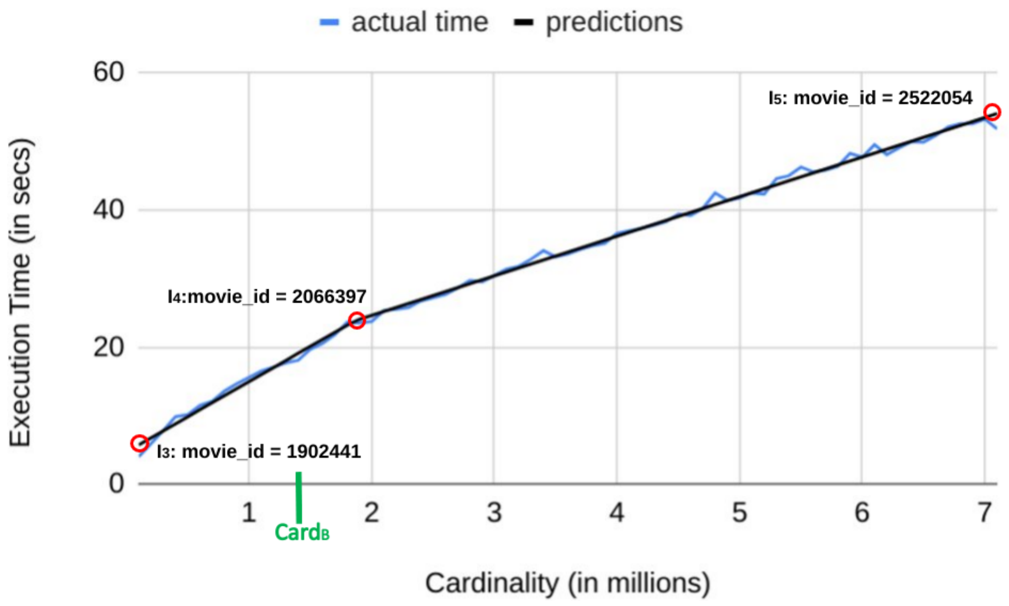
(a) Curve 1



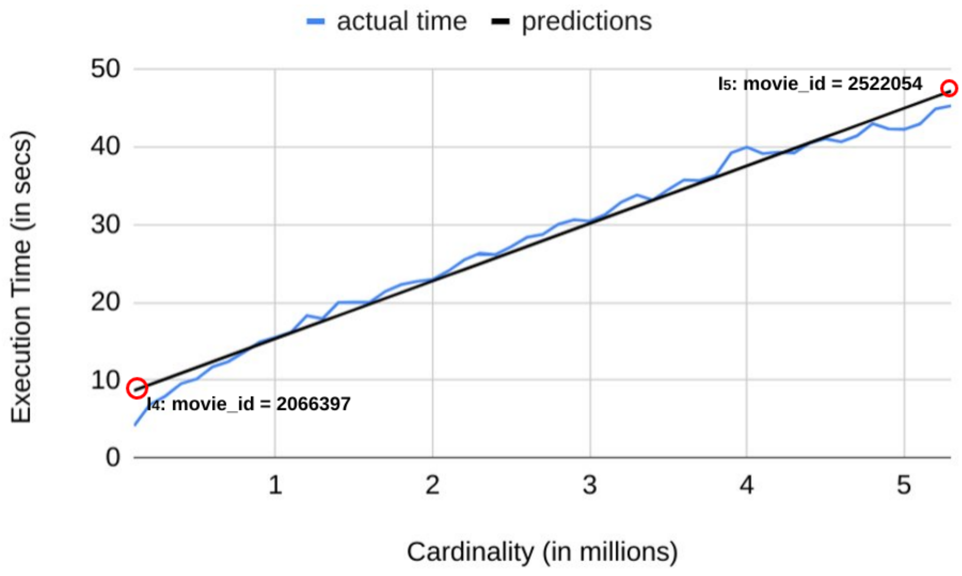
(b) Curve 2

Figure 3.3: Training Curves 1 and 2 for Prediction of Range Index Scan

3. Repeat step 2 to get the learned curves 3 and 4 (Figures 3.4a and 3.4b) until no new internal inflexion points are required. At point $I_4$, again there was an increase of correlation from -0.02 to 0.03 and around 20% of data pages accessed were already found cached. The requirement of an inflexion point is decided based on training error. If error on each point



(a) Curve 3



(b) Curve 4

Figure 3.4: Training Curves 3 and 4 for Prediction of Range Index Scan

of the training curve is less than a pre-determined threshold (5 seconds here), then we need only one line (i.e. zero internal inflexion points) for fitting. The final set of inflexion points is the union of inflexion points in all curves. While generating a new curve at any step, most of the inflexion points either repeat from the old set or lie very close. The inflexion points that lie close can be clubbed into one by taking the average. Note that in our example, we have got five inflexion points, $I_1$ to $I_5$.

#### 3.2.2.2 Prediction Procedure

Suppose we want to predict execution time of the query:

$$select\ *\ from\ movie\_info\ where\ A \leq movie\_id \leq B$$

where A and B are constants. Let's call the curves got in the training phase as $C_1$, $C_2$, ...., $C_m$ ($m = 4$ in our example) and the inflexion points as $I_1$, $I_2$,..., $I_n$ ($n = 5$ in our example). First, the points A and B are mapped to their respective cardinalities, which we learnt using piecewise linear approximation as well. Let's call the cardinalities w.r.t. attribute values A and B as $card_A$ and $card_B$. Suppose the range $[card_A, card_B]$ lies between $I_1$ and $I_4$, as shown in Figure 3.4. Then the range $[card_A, card_B]$ is partitioned by the inflexion points into three partitions - $[card_A, I_2]$, $[I_2, I_3]$ and $[I_3, card_B]$. We need to predict time spent in each of these partitions. Let's call the contribution by curve $C_k$ for a partition $[i, j]$ as $C_k(i, j)$, where $C_k(i, j) = C_k(I_k, j) - C_k(I_k, i - 1)$ where $I_k$ is the first inflexion point in the curve $C_k$. One way to predict total execution time for range $[card_A, card_B]$ is to simply add $C_1(card_A, I_2)$, $C_2(I_2, I_3)$ and $C_3(I_3, card_B)$. But this way, we are ignoring the caching effect of range $[card_A, I_2]$ on range $[I_2, I_3]$ and of range $[card_A, I_3]$ on range $[I_3, card_B]$. To tackle this problem, we apply a heuristic of averaging the time across all curves for every partition. For example, for partition $[I_2, I_3]$, we average time predicted by Curve 1 and Curve 2 which, for this particular partition, consider and do not consider the caching effect of range $[card_A, I_2]$ respectively. Then the predicted execution time for the aforementioned query will be given by:

$$Predicted\ time = C_1(card_A, I_2)\ + avg(C_1(I_2, I_3), C_2(I_2, I_3)) + \\ avg(C_1(I_3, card_B),\ C_2(I_3, card_B),\ C_3(I_3, card_B))$$

A simple case would be when the range $[card_A, card_B]$ lies in, say the range $[I_2, I_3]$. Then, predicted time would simply be $C_2(card_A, card_B)$. Note that the first curve to contribute to the predicted time is the one for which there is no internal inflexion point before $card_A$. The rest of the curves are simply the ones which contain a portion of the range. The intuition behind the model is that the inflexion points capture the change in increase of execution time due to factors like fragmentation or caching or correlation effect. The predictions of the queries in Figure 3.2 using our model were 4s, 21s and 64s respectively.

# Chapter 4

# Prediction of Sort Operator

```
SELECT *
FROM R
ORDER BY A1, A2, .. , Ak
```

Figure 4.1: Sort Query Template $Q\langle R, \{A1, A2, .., Ak\}\rangle$

Given a query $Q$ (Figure 4.1), Postgres uses quicksort to sort the relation $R$ on the ordered set of k attributes $\{A1, A2, \ldots, Ak\}$.

## 4.1 PostgreSQL's Cost Model for Sort

Assuming the cardinality of relation $R$ is $n$, Postgres charges an expected comparison cost of $\theta(nlogn)$ (multiplied with $2 * CPU\_OP\_COST$ - a heuristic for a single unit of comparison cost). It also charges an output processing cost of $CPU\_OP\_COST$ per tuple, which is the case for all operators as we will see.
Thus, Postgres' cost formula for sorting is as follows:

$$Comparison\ Cost = 2\ *\ CPU\_OP\_COST\ * input\_card\ *\ log_2(input\_card)$$

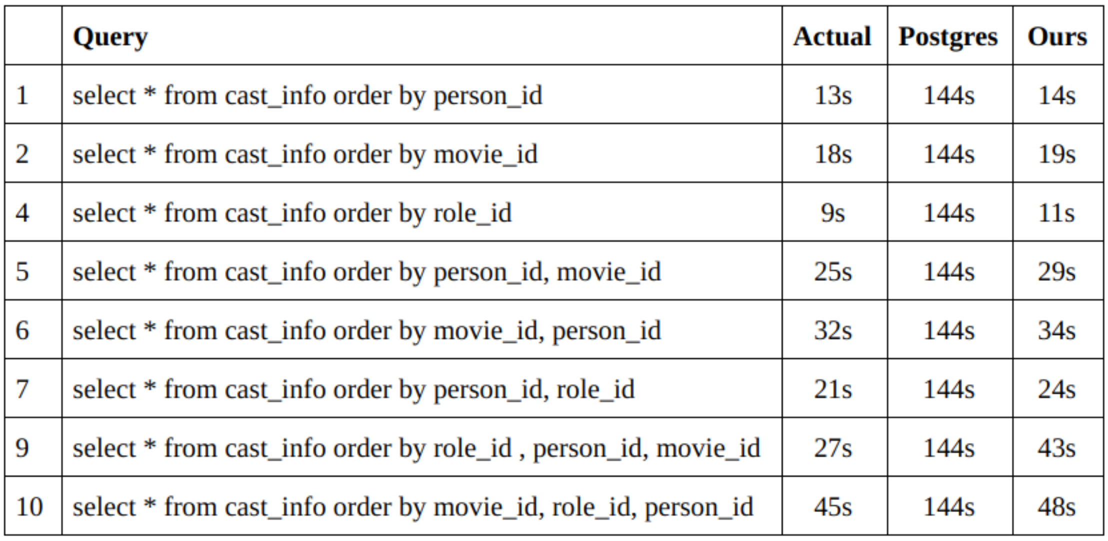
$$Output\ Cost = CPU\_OP\_COST\ *\ input\_card$$

$$Final\ Cost = Comparison\ Cost + Output\ Cost$$

### 4.1.1 Shortcomings

The caveats in the Postgres' model are as follows:

1. The execution time of sorting a relation depends not only on the number of comparisons but also on the actual movement of tuples. The latter is dependent primarily on two data



| | Query | Actual | Postgres | Ours |
|---|---|---|---|---|
| 1 | select * from cast_info order by person_id | 13s | 144s | 14s |
| 2 | select * from cast_info order by movie_id | 18s | 144s | 19s |
| 4 | select * from cast_info order by role_id | 9s | 144s | 11s |
| 5 | select * from cast_info order by person_id, movie_id | 25s | 144s | 29s |
| 6 | select * from cast_info order by movie_id, person_id | 32s | 144s | 34s |
| 7 | select * from cast_info order by person_id, role_id | 21s | 144s | 24s |
| 9 | select * from cast_info order by role_id , person_id, movie_id | 27s | 144s | 43s |
| 10 | select * from cast_info order by movie_id, role_id, person_id | 45s | 144s | 48s |

Figure 4.2: Sort Operator Examples

properties: pre-sortedness and duplication factor. For example, a pre-sorted array will incur zero tuple movement cost, and an array that is not pre-sorted but has large number of duplicates (for instance, an array of only two distinct values occurring consecutively) will also incur very less movement cost. Though the duplication factor can be seen as a specific case of pre-sortedness, we see these two differently because of the way we measure them. For a single attribute, we measure pre-sortedness as the (absolute value of) correlation coefficient (corr) between an array and its sorted counterpart, and duplication factor (dup) as:

$$dup(\#rows, \#distinct\_values) = \frac{\#rows - \#distinct\_values}{\#rows - 1}$$

2. Postgres does not take into consideration the number of attributes ($k$) in cost estimation. Thus, for any number of attributes, Postgres' estimated cost of the sort query remains same. For example, in Figure 4.2, we ran a couple of queries on the same relation $cast\_info$ but different ORDER BY clauses. But Postgres predicted same cost of all queries no matter how many or what attributes there were in the Order By clause.

3. As a result of above, Postgres ignores the order of attributes as well. It's important to consider the order because, for instance, if the first attribute has all distinct values, then the subsequent attributes will not even be considered for comparison. On the other hand, if the first attribute has all elements same, then the next attribute will decide the sorting order and thus add to the sorting cost.

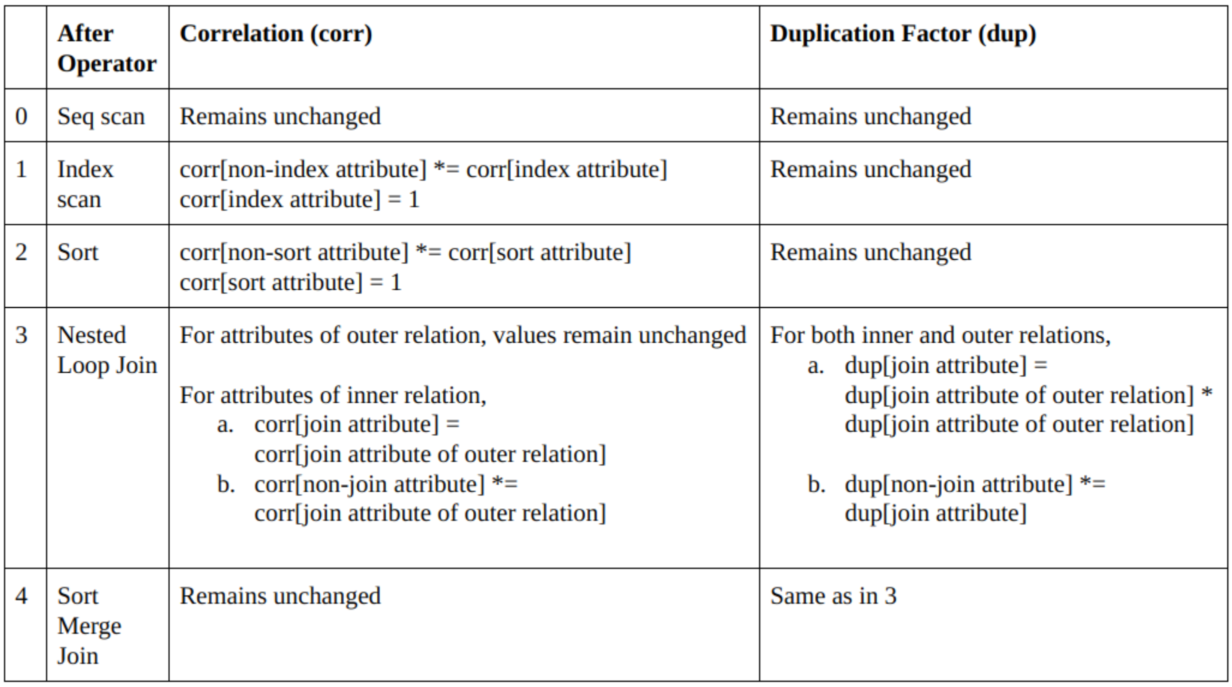
## 4.2 Our Cost Model for Sort

**Algorithm 1:** For prediction of sorting cost of an ordered set of attributes

```
Function Cost_Sort_Single_Attr
    Input: corr, dup, input_card
    Output: sorting cost of a single attribute
    $comparison\ cost = COMPARISON\_COST * input\_card * log_2(input\_card)$;
    $movement\ cost = (1 - (W * corr + (1 - W) * dup)) * MOVEMENT\_COST * input\_card * log_2(input\_card)$;
    return $comparison\ cost + movement\ cost$;

Function Cost_Sort
    Input: num_attributes, corr[], dup[], input_card
    Output: sorting cost of an ordered set of attributes
    $total\_cost = 0$;
    $current\_dup = 1$;
    $current\_corr = 1$;
    for $i = 0;\ i < num\_cols;\ i++$ do
        $total\_cost += current\_dup * Cost\_Sort\_Single\_Attr\ (current\_corr * corr[i], dup[i], input\_card)$;
        $current\_corr *= corr[i]$;
    end
    $output\_cost = CPU\_TUPLE\_COST * input\_card$;
    return $total\_cost + output\_cost$;
```

There are three major costs involved in sorting - *comparison cost*, *movement cost* and *output cost*. We introduce two new local cost parameters - $COMPARISON\_COST$ and $MOVEMENT\_COST$. The former is tuned on an attribute with no expected movement (i.e. $corr = 1$) and the latter is tuned on an attribute with maximum movement (i.e. $corr = 0$, $dup = 0$). The $COMPARISON\_COST$ was found to be around eleven times lower and the $MOVEMENT\_COST$ around four times lower than the $CPU\_OP\_COST$ on our machine. For example, for first query in Figure 4.2, Postgres estimated 144s for sorting on attribute $person\_id$ of relation $cast\_info$ when in fact, the actual time was very less (13s). Our model predicted a comparison cost of 6s and a movement cost of 7s ($corr = 0.7$, $dup = 0.8$). Similarly,



| | After Operator | Correlation (corr) | Duplication Factor (dup) |
|---|---|---|---|
| 0 | Seq scan | Remains unchanged | Remains unchanged |
| 1 | Index scan | corr[non-index attribute] *= corr[index attribute]<br>corr[index attribute] = 1 | Remains unchanged |
| 2 | Sort | corr[non-sort attribute] *= corr[sort attribute]<br>corr[sort attribute] = 1 | Remains unchanged |
| 3 | Nested Loop Join | For attributes of outer relation, values remain unchanged<br><br>For attributes of inner relation,<br>a. corr[join attribute] = corr[join attribute of outer relation]<br>b. corr[non-join attribute] *= corr[join attribute of outer relation] | For both inner and outer relations,<br>a. dup[join attribute] = dup[join attribute of outer relation] * dup[join attribute of outer relation]<br><br>b. dup[non-join attribute] *= dup[join attribute] |
| 4 | Sort Merge Join | Remains unchanged | Same as in 3 |

Figure 4.3: Heuristics for change in corr and dup values of an attribute in a plan

our model predicted a comparison cost of 6s for both second and third query but a movement cost of 13s for *movie_id* ($corr = 0.005$, $dup = 0.9$) and 5s for *role_id* ($corr = 1$, $dup = 0.9$).

In case of multiple attributes, the first attribute ($A1$) will contribute all the sorting time that it would if it were in isolation. The contribution of the second attribute ($A2$) is dependent on $dup[A1]$. If $dup[A1] = 1$, $A2$ will also contribute all the sorting time that it would if it were in isolation. But if $dup[A1] = 0$, this means $A1$ has all distinct elements and thus, there won't be any contribution to sorting time from $A2$ to $Ak$. We apply the heuristic of multiplying the sorting time of $A2$ with $dup[A1]$ to get its contribution. Similarly with the subsequent attributes. Note that we are assuming attribute value independence here. Algorithm 1 shows the prediction algorithm in which the main function *Cost_Sort* calls *Cost_Sort_Single_Attr* to calculate the contribution of every attribute in the Order By clause. Note that in the *movement_cost*, W is a weight parameter discovered during tuning that decides how much weight is given to corr and how much to dup. In our experiments, W was set to 0.4.

The corr and dup values for all attributes of all relations are kept pre-computed in the pre-compilation phase. However, the values of the intermediate relations that a plan may generate are unknown beforehand. In Figure 4.3, we came up with a set of simple rules on how corr and dup values can change when a certain operation occurs. The rules have been kept simple because it is difficult to calculate the exact values of corr and dup for intermediate relations without keeping significant amount of metadata. The assumption of attribute value independence holds here as well. These formulae are w.r.t. a single attribute. The formulae for the case of multiple attributes are easily derivable.

# Chapter 5

# Prediction of Join Operators

```
SELECT *
FROM R1, R2
WHERE R1.A1=R2.B1 AND .. AND R1.Ak=R2.Bk
```

Figure 5.1: Join Query Template $Q\langle R1, R2, \{\{A1, B1\}, .., \{Ak, Bk\}\}\rangle$

Given a query Q (Figure 5.1) with two input relations R1 and R2 and a number of join predicates (of type $R1.Ai = R2.Bi$), Postgres can choose any join operator among Nested Loop Join (NLJ), Sort Merge Join (SMJ) and Hash Join to execute the join operation. In this work, we only talk about prediction of NLJ and SMJ operators in detail below.

## 5.1 Nested Loop Join

Given a query Q (Figure 5.1), Postgres executes the NLJ operator as follows - for each tuple scanned from the outer relation, Postgres loops through the inner relation and checks for the tuples that join with the tuple from outer relation based on the predicate conditions. However, there are a few exceptions to this execution behaviour:

- Case 1: When the joining attribute(s) of inner relation is known to be unique (for instance, in presence of PRIMARY KEY or UNIQUE constraint), for every outer relation tuple, the loop on the inner relation stops if and when the unique value is found.
- Case 2: If the right child of NLJ is an index scan, only the tuples of the inner relation corresponding to each outer tuple are scanned using index.

### 5.1.1 PostgreSQL's Cost Model for Nested Loop Join

Postgres charges a processing cost and a predicate evaluation cost for NLJ. Another cost called rescan cost is charged for scanning the tuples from the relation present in memory after being scanned from disk. It is called rescan cost to differentiate from the first scan that happens from disk.

$$Rescan\ Cost = CPU\_OP\_COST\ * inner\_rel\_card\ *\ outer\_rel\_card$$

$$Processing\ Cost = CPU\_TUP\_COST\ * inner\_rel\_card\ * outer\_rel\_card$$

$$Predicate\ Evaluation\ Cost = CPU\_OP\_COST\ * \#join\_predicates\ *\ inner\_rel\_card\ * \\ outer\_rel\_card$$

$$Final\ Cost = Rescan\ Cost + Processing\ Cost + Predicate\ Evaluation\ Cost$$

For case 1, the expected number of probes in inner relation for each "matched" outer relation tuple is half the total size of the inner relation while for an "unmatched" tuple, the whole inner relation is probed. For case 2, Postgres uses the same model that it uses for index scan in isolation with a few minor changes.

### 5.1.2 Our Cost Model for Nested Loop Join

Postgres was found to overestimate the execution time of NLJ in all the cases. We have replaced the global cost parameters with local parameters in the above cost functions as follows:

$$Rescan\ Cost = CPU\_RESCAN\_COST\ *\ inner\_card\ *\ outer\_card$$

$$Processing\ Cost = NLJ\_PROCESSING\_COST\ *\ inner\_rel\_card\ *\ outer\_rel\_card$$

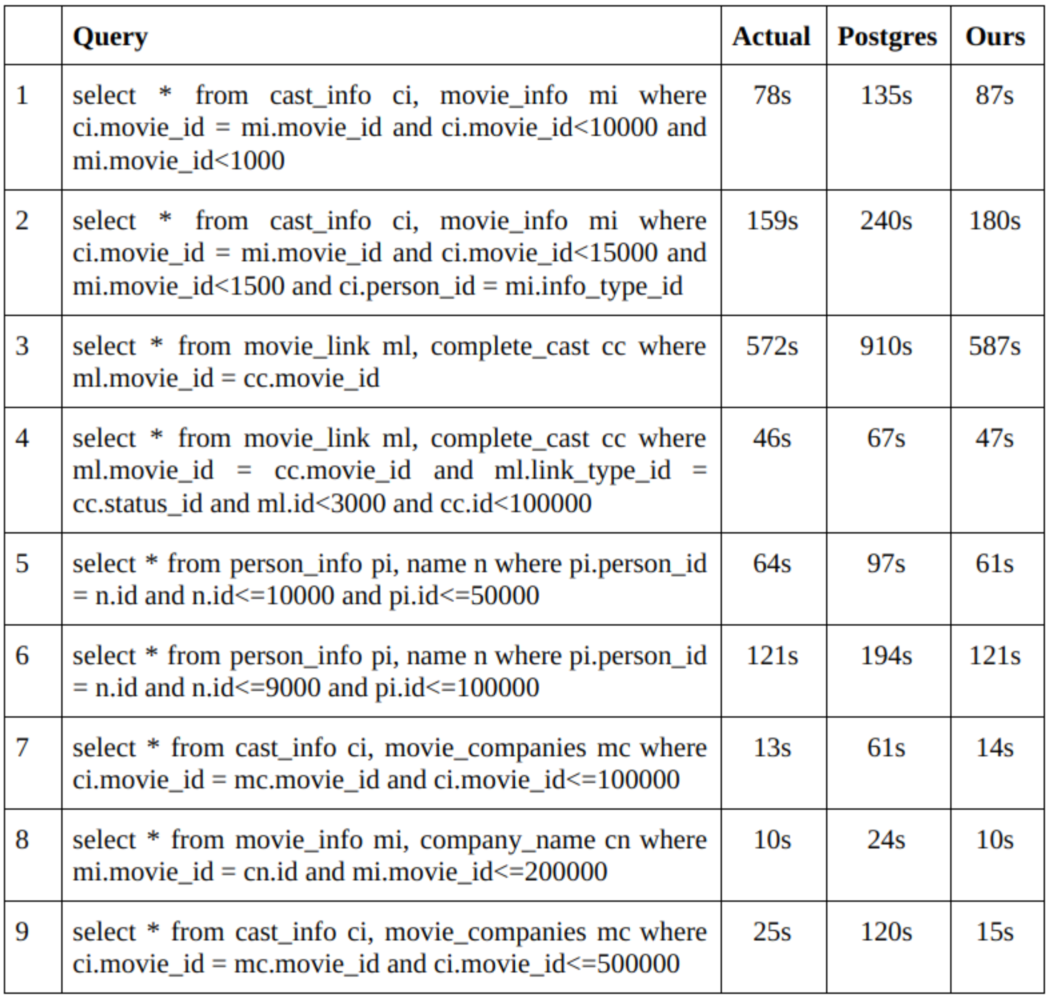
$$Predicate\ Evaluation\ Cost =\ JOIN\_PRED\_EVAL\_COST\ *\ \#join\_predicates\ * \\ inner\_rel\_card\ *\ outer\_rel\_card$$

$$Output\ Cost = CPU\_TUP\_COST\ *\ output\_card$$

$$Final\ Cost = Rescan\ Cost + Processing\ Cost + Predicate\ Evaluation\ Cost + Output\ Cost$$

The cost parameters $CPU\_RESCAN\_COST$ and $JOIN\_PRED\_EVAL\_COST$ were found to be around two times and six times lower than $CPU\_OP\_COST$ respectively after tuning and $NLJ\_PROCESSING\_COST$ was found to be a little lower than $CPU\_TUP\_COST$. $CPU\_RESCAN\_COST$ can be tuned using any NLJ query because the rescanning cost is explicitly shown in PostgreSQL's output query plan. $NLJ\_PROCESSING\_COST$ can be tuned by running an NLJ query with zero predicates and $JOIN\_PRED\_EVAL\_COST$ can be tuned using a single predicate and then subtracting the costs of other NLJ cost functions.

We handle Case 1 like Postgres does, because in expectation (as confirmed experimentally), for each matched outer tuple, the fraction of inner table scanned is usually half. Coming to Case 2, since Postgres' model of this case is just an extension of index scan in isolation, the problems remain the same. However, single index scan is one-time and join index-scan is repetitive, and therefore brings in more caching effects. So to model this case, we again use the power of machine learning.



| | Query | Actual | Postgres | Ours |
|---|---|---|---|---|
| 1 | select * from cast_info ci, movie_info mi where ci.movie_id = mi.movie_id and ci.movie_id<10000 and mi.movie_id<1000 | 78s | 135s | 87s |
| 2 | select * from cast_info ci, movie_info mi where ci.movie_id = mi.movie_id and ci.movie_id<15000 and mi.movie_id<1500 and ci.person_id = mi.info_type_id | 159s | 240s | 180s |
| 3 | select * from movie_link ml, complete_cast cc where ml.movie_id = cc.movie_id | 572s | 910s | 587s |
| 4 | select * from movie_link ml, complete_cast cc where ml.movie_id = cc.movie_id and ml.link_type_id = cc.status_id and ml.id<3000 and cc.id<100000 | 46s | 67s | 47s |
| 5 | select * from person_info pi, name n where pi.person_id = n.id and n.id<=10000 and pi.id<=50000 | 64s | 97s | 61s |
| 6 | select * from person_info pi, name n where pi.person_id = n.id and n.id<=9000 and pi.id<=100000 | 121s | 194s | 121s |
| 7 | select * from cast_info ci, movie_companies mc where ci.movie_id = mc.movie_id and ci.movie_id<=100000 | 13s | 61s | 14s |
| 8 | select * from movie_info mi, company_name cn where mi.movie_id = cn.id and mi.movie_id<=200000 | 10s | 24s | 10s |
| 9 | select * from cast_info ci, movie_companies mc where ci.movie_id = mc.movie_id and ci.movie_id<=500000 | 25s | 120s | 15s |

Figure 5.2: Nested Loop Join Operator Examples

Note that Case 2 is a much harder problem than single index scan and cannot be dealt with the way we modelled single index scan - i.e. pre-processing on the data layout of relations because then we will have to pre-process on each possible stream of numbers (which are exponential in number) unlike single index scan (in which case we had to pre-process only on the stream of attribute values in ascending order). Instead, we have tried learning a piecewise linear approximation curve which predicts execution time of scanning the right relation for a given cardinality of distinct values from the left relation. For each index attribute on each relation, the training data was created by randomly sampling a fixed number, say t, of instances of k random unique values in the domain of the index attribute. Here, k is varied from minimum to maximum domain value of the index attribute in fixed intervals. So, for k unique values coming from the left child, we predict execution time of scanning those k unique values in the right child relation based on the t instances available. However, note that we do not know the number of distinct values coming from the left child of the NLJ operator. So, for this, we use the dup factor (as computed in Figure 4.3) on the left child's joining attribute.

In Figure 5.2, we have shown examples of a couple of queries on the IMDB database. The queries 1 to 4 are for the regular case, queries 5 and 6 are for case 1 and queries 7 to 9 are for case 2. We talk about more extensive evaluation in Chapter 7.

## 5.2 Sort Merge Join

For the same join query in Figure 5.1, SMJ operator first sorts the two input relations on the joining attributes and then merges them using the merge operation. In presence of duplicates in the outer relation, the corresponding tuples in the inner relation could be scanned multiple times. The output cardinality of the join can be seen as a reasonable measure of the number of tuples rescanned from the inner relation.

### 5.2.1 PostgreSQL's Cost Model for Sort Merge Join

Just like in NLJ, Postgres charges a *Processing Cost*, *Predicate Evaluation Cost* and *Output Cost* here as well. The *Rescan Cost* of tuples from inner relation are accounted for in the *Processing Cost* itself. Note that all these cost functions use the same global cost parameter $CPU\_OP\_COST$.

$$Processing\ Cost = CPU\_OP\_COST\ *\ (outer\_rel\_card + output\_card)$$

$$Predicate\ Evaluation\ Cost = CPU\_OP\_COST\ * \#join\_predicates\ *\ output\_card$$

$$Output\ Cost = CPU\_OP\_COST\ *\ output\_card$$

$$Final\ Cost = Processing\ Cost + Predicate\ Evaluation\ Cost + Output\ Cost$$

### 5.2.2 Our Cost Model for Sort Merge Join

Like NLJ, we have replaced the global cost parameters in the cost functions with new local cost parameters in the light of few limitations in Postgres' model, as follows:

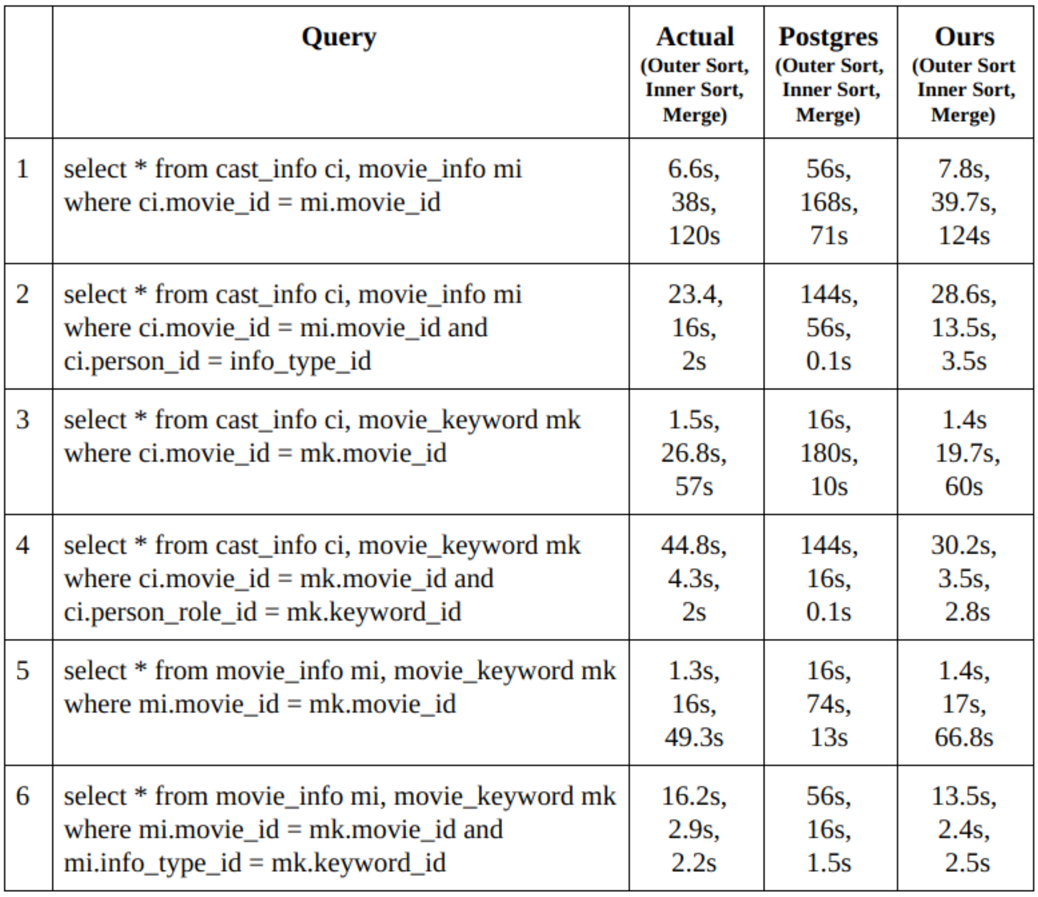
$$Rescan\ Cost = CPU\_RESCAN\_COST\ *\ max(output\_card\ -\ inner\_card, 0)$$

$$Processing\ Cost =\ SMJ\_PROCESSING\_COST\ *\ output\_card$$

$$Predicate\ Evaluation\ Cost = \ JOIN\_PRED\_EVAL\_COST * \#join\_predicates * output\_card$$

$$Output\ Cost = CPU\_TUP\_COST\ *\ output\_card$$

$$Final\ Cost = Rescan\ Cost + Processing\ Cost + Predicate\ Evaluation\ Cost + Output\ Cost$$



| | Query | Actual (Outer Sort, Inner Sort, Merge) | Postgres (Outer Sort, Inner Sort, Merge) | Ours (Outer Sort Inner Sort, Merge) |
|---|---|---|---|---|
| 1 | select * from cast_info ci, movie_info mi where ci.movie_id = mi.movie_id | 6.6s, 38s, 120s | 56s, 168s, 71s | 7.8s, 39.7s, 124s |
| 2 | select * from cast_info ci, movie_info mi where ci.movie_id = mi.movie_id and ci.person_id = info_type_id | 23.4, 16s, 2s | 144s, 56s, 0.1s | 28.6s, 13.5s, 3.5s |
| 3 | select * from cast_info ci, movie_keyword mk where ci.movie_id = mk.movie_id | 1.5s, 26.8s, 57s | 16s, 180s, 10s | 1.4s 19.7s, 60s |
| 4 | select * from cast_info ci, movie_keyword mk where ci.movie_id = mk.movie_id and ci.person_role_id = mk.keyword_id | 44.8s, 4.3s, 2s | 144s, 16s, 0.1s | 30.2s, 3.5s, 2.8s |
| 5 | select * from movie_info mi, movie_keyword mk where mi.movie_id = mk.movie_id | 1.3s, 16s, 49.3s | 16s, 74s, 13s | 1.4s, 17s, 66.8s |
| 6 | select * from movie_info mi, movie_keyword mk where mi.movie_id = mk.movie_id and mi.info_type_id = mk.keyword_id | 16.2s, 2.9s, 2.2s | 56s, 16s, 1.5s | 13.5s, 2.4s, 2.5s |

Figure 5.3: Sort Merge Join Operator Examples

Like in NLJ, we replaced the global cost parameter $CPU\_OP\_COST$ with local parameters $SMJ\_PROCESSING\_COST$ and $JOIN\_PRED\_EVAL\_COST$ in $Processing\ Cost$ and $Predicate\ Evaluation\ Cost$ respectively, which are easily tunable using two SMJ queries, former with zero predicates and zero duplicates in outer relation and latter with an SMJ query with one predicate and zero duplicates. Further, we broke Postgres' $Processing\ Cost$ formula into $Rescan\ Cost$ and $Processing\ Cost$ to tune a new local parameter $RESCAN\_COST$ separately.

In Figure 5.3, we show a few example queries executed on IMDB dataset, along with the actual execution times, Postgres' estimated times and our predicted times for the outer sort operation, inner sort (+ rescan) operation and the merge operation.

# Chapter 6

# Prediction of Group By Operator with Aggregation

```
SELECT agg(B1), agg(B2),.., agg(Bm)
FROM R
GROUP BY A1, A2, .., Ak
```

Figure 6.1: Group By with Aggregate Query Template $Q\langle R, \{A1, A2, .., Ak\}, \{B1, B2, .., Bm\}\rangle$

Given an aggregate query Q (Figure 6.1) which groups the relation R on an ordered set of attributes $\{A1, A2, ..., Ak\}$ and applies aggregation function (agg = count/sum/avg/min/max) on attributes $B1, B2, ..., Bm$, Postgres can implement the Group By with Aggregate operation in either of the two following ways:

- SORT-AGGREGATE: This method first sorts the relation on the Group By attributes and then applies aggregation. Aggregation takes one pass through the relation and finds values of each aggregate function for each group. Note that for each tuple, a comparison is done with the previous tuple to check if it belongs to the previous group or it's a new group.
- HASH-AGGREGATE: This method first inserts the tuples of the relation into a hash table based on the hashing of the Group By attributes and then applies aggregation for each hash bucket. Note that since there can be collisions in the hash bucket, comparisons are required here as well.

Postgres uses the same model for both of the above scenarios, which is as follows:

$$Group\ Comparison\ Cost\ =\ CPU\_OP\_COST\ \ *\ num\_group\_by\_columns\ \ *\ \ input\_card$$

$$Aggregation\ \ Cost\ \ =\ \ CPU\_OP\_COST\ \ \ *\ num\_aggregation\_columns\ \ \ *\ \ \ input\_card$$

$$Output\ Cost = CPU\_TUP\_COST\ \ *\ num\_groups$$

$$Final\ Cost = Group\ Comparison\ Cost + Aggregation\ Cost + Output\ Cost$$

Since JOB queries do not have Group By clause, we created a new set of queries by adding Group By clauses to JOB queries and tested Postgres' model on it. The estimations were very close to accurate on the queries, incurring a relative error of less than 10% on most of them. Thus, we use PostgreSQL's model as it is.

# Chapter 7

# Experiments

In this chapter, we evaluate our cost model against a well-tuned PostgreSQL's cost model. To reiterate, we have taken the following assumptions for our execution environment: a) Single query and isolated environment (i.e. no other user processes running on the system), b) Disk-resident database, c) Cold cache (i.e. memory is cleared before running a query), d) Indexes built only on numerical attributes and d) Perfect cardinality estimates. All experiments are run on an Intel Core i9-7900X machine with 3.30GHz CPU, 32GB RAM, 2TB 7200RPM HDD and 64-bit Ubuntu 18.04. In this chapter, we consider the no spilling scenario and defer its discussion to the next chapter.

## 7.1 Evaluation Benchmark

We evaluate our model on the Join Order Benchmark (JOB [6]). However, we do not evaluate on the original query plans we got by executing JOB queries on PostgreSQL's native optimiser. Rather, we have created a host of six different suites of query plans using JOB queries for training purposes because of class bias in the operator frequencies of the original plans. We forced different combinations of possible choices of operators (by changing the configuration parameters of PostgreSQL) and came up with the following suites (each query plan has scan operators, join operators and an aggregate operator):

- Suite 1: Scan: Sequential, Join: NLJ
- Suite 2: Scan: Sequential, Join: SMJ
- Suite 3: Scan: Sequential (all filters removed), Join: SMJ
- Suite 4: Scan: Sequential, Join: Any

- Suite 5: Scan: Index, Join: SMJ
- Suite 6: Scan: Any, Join: Any

The Suite 1 forces Sequential scan and NLJ operator on the optimiser. The execution times of query plans in this suite range from a 5 secs to a 5 hours. The Suite 2 forces Sequential scan and SMJ operator. The execution times of these query plans are all under 1 min. Thus we created another suite (Suite 3) by removing all scan filters to test our model on high execution times as well. Suite 4 forces only Sequential scan and leaves it on the native optimiser to pick from NLJ and SMJ operators for join operation (the presence of these two operators in the plans we got is almost equal). This way, we can do evaluation on interaction between the two join operators. Suite 5 forces Index scan and SMJ operator and Suite 6 does not force any operator. These two suites are for evaluation of index scan in isolation (as child nodes of SMJ and left child of NLJ) and as right child of NLJ. We did not create a separate suite by forcing Index Scan and NLJ because in Suite 6, most of the join operators were NLJ with index scan, thus avoiding redundancy. And we did not create another suite by forcing Index Scan with Any Join operator because we are already testing interaction between NLJ and MJ in Suite 4. JOB queries do not have GROUP BY clause by default and we did not add them in these suites so as to focus on the errors of only error-prone operators. We have further made sure that all plans are unique.

## 7.2 Performance Evaluation

For each query plan, we measure the Qerror for PostgreSQL's model and our model, where Qerror(actual execution time($time_{act}$), estimated execution time($time_{est}$)) is defined as:

$$Qerror(time_{act}, time_{est}) = max( \frac{time_{act}}{time_{est}}, \frac{time_{est}}{time_{act}} )$$

For any query plan, the goal is to get a Qerror of as close as possible to 1. Note that since Qerror is a multiplicative error, Qerror of, say 2 for a plan with execution time 10s means an estimated time of 5s or 20s but for a plan with execution time 60 mins, it means an estimation of 30 mins or 120 mins. Thus, the former estimation is not so bad in comparison to the latter. For all the graphs below, the query plans have been ordered in the increasing order of execution times from left to right. For each suite, we use the vector [mean Qerror, median Qerror, max Qerror] as the evaluation metric ($P_Q$ for Postgres and $O_Q$ for ours). Further, for each suite, we show Qerrors on per-query basis in bar graphs as well. As visible in graphs of all suites (Figures

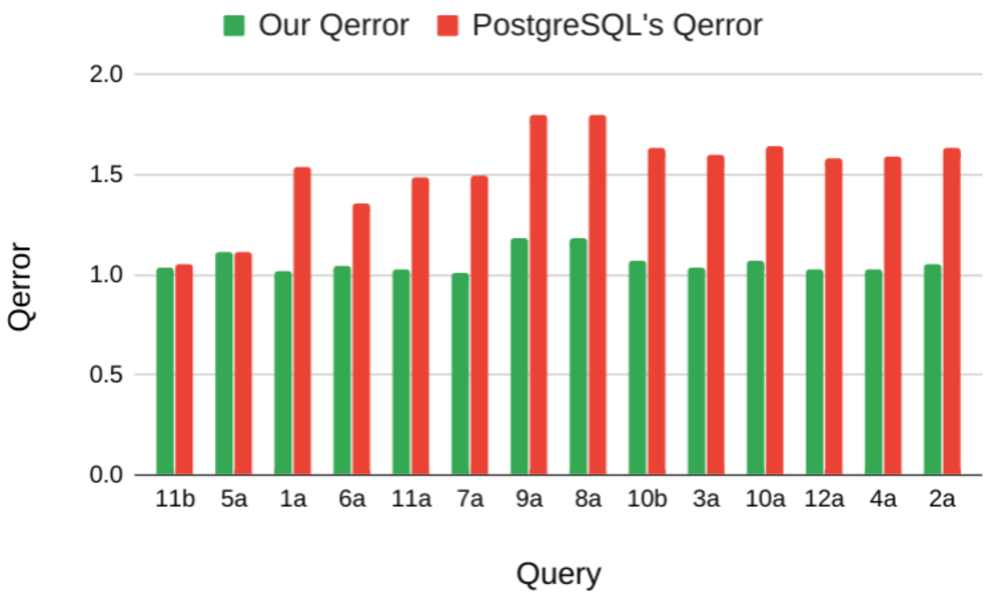


Figure 7.1: Performance on Suite 1

7.1 to 7.6), for each single query, we estimate better than Postgres. For every suite, we have removed the outliers. Keeping this in mind, the evaluation on the suites is as follows:

**Suite 1:** The execution times of these 15 plans (Figure 7.1) range from 5s to 5 hours. First 12 plans are under 10 mins. Last 3 plans took 24 mins, 25 mins and 5 hours respectively. Here, $P_Q : [1.5, 1.6, 1.8], O_Q : [1.06, 1.04, 1.18]$. Postgres overestimated the execution times of all of these plans. For example, for the plan 4a, the actual time was 25 mins. Postgres predicted it to be 40 mins but by simply tuning the local cost parameter of NLJ, we brought the prediction to 25.5 mins.

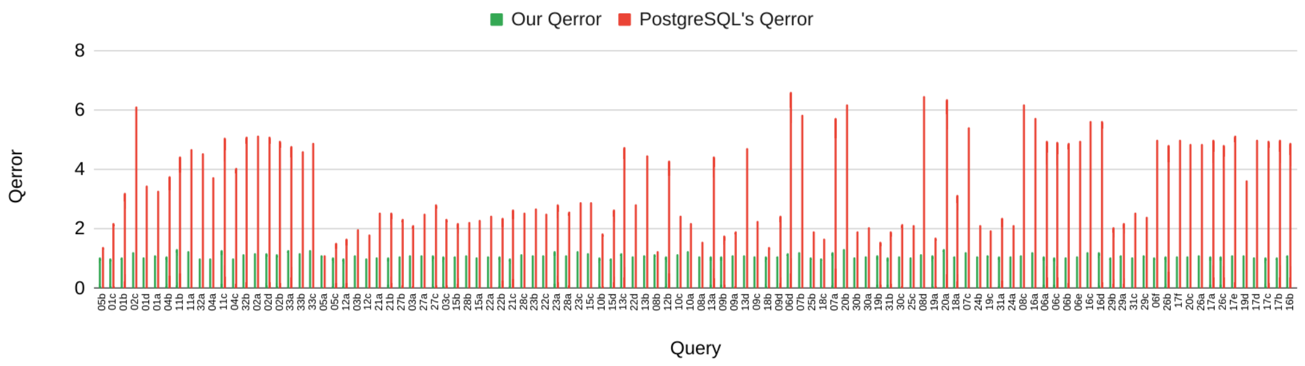


Figure 7.2: Performance on Suite 2

**Suite 2:** The execution times of these 108 plans range from 3s to 47s in the Figure 7.2. Here, $P_Q = [3.5, 2.8, 6.6]$ and $O_Q = [1.1, 1.1, 1.3]$. Postgres overestimated the times of all these plans mostly because of overestimations on the sort operator. For example, for the plan 19d, the actual time was 46s, but Postgres estimated it to be 3 mins and we predicted 52s for it.

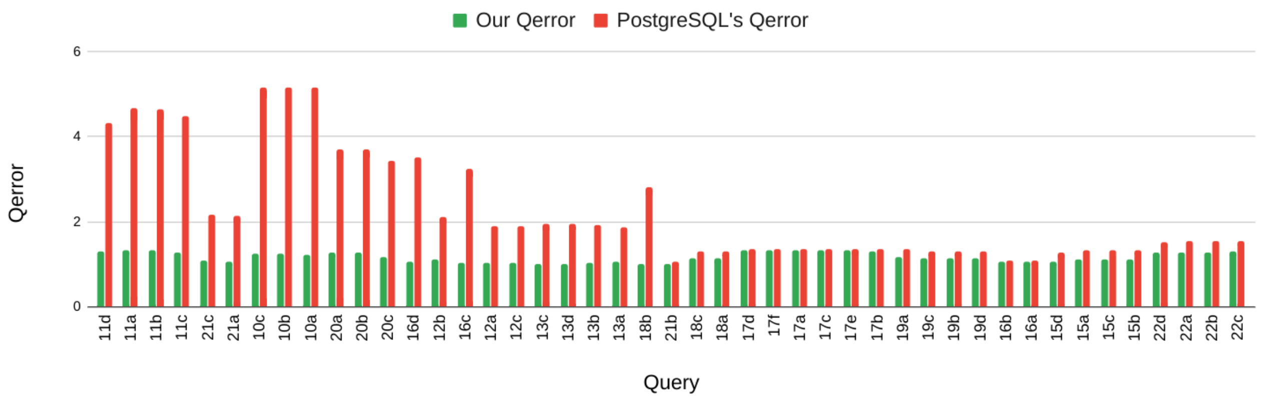


Figure 7.3: Performance on Suite 3

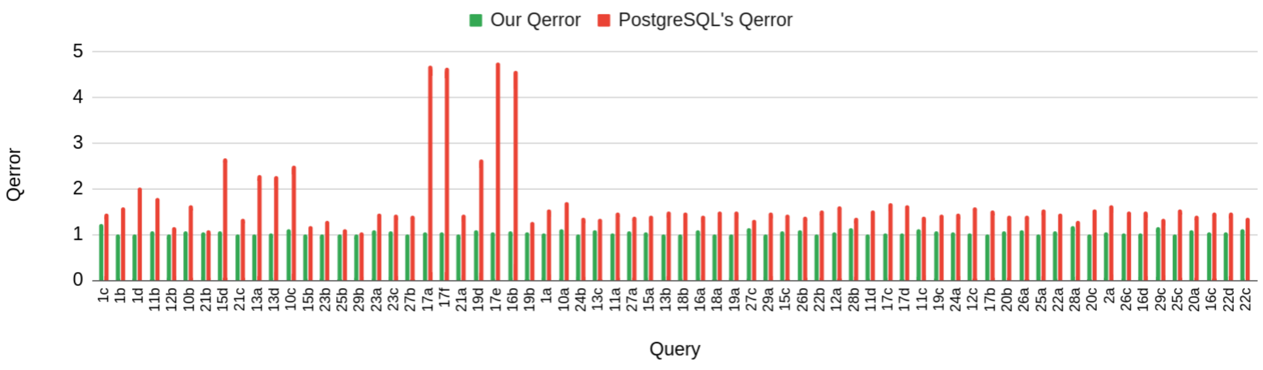


Figure 7.4: Performance on Suite 4

**Suite 3:** The execution times of these 49 plans (Figure 7.3) range from 11s to 2.5 hours. The execution times till the 18b query plan are under 3 mins. From the next plan onwards, the execution times are more than 7 hours. Note that for 7 hours, even a Qerror of 1.1 means a difference of 40 mins. Here, $P_Q : [2.3, 1.5, 5.1], O_Q : [1.2, 1.1, 1.3]$. For all these plans, again Postgres overestimated the times, same as in Suite 2. For example, for plan 18a, the actual time was 11 mins which we predicted as 10 mins but Postgres estimated as 15 mins.

**Suite 4:** The execution times of these 70 plans (Figure 7.4) range from 4s to 12 hours. First 28 plans (till 1a) have execution times under 1 min. Next 12 plans (till 24b) have times under 5 mins. Next 6 plans have time under 1 hour and then the execution times increases gradually. Here, $P_Q : [1.7, 1.5, 4.7], O_Q : [1.05, 1.04, 1.2]$. The proportion of NLJ and MJ in these plans is almost equal. So, Postgres' high Qerrors on MJ are compensated for low Qerrors on NLJ. But, since the times of some plans are high, even low Qerrors mean grave differences. For example, for plan 22d, the actual time was 10.5 hours whereas Postgres predicted 16 hour for it and we predicted 10 hours for it.

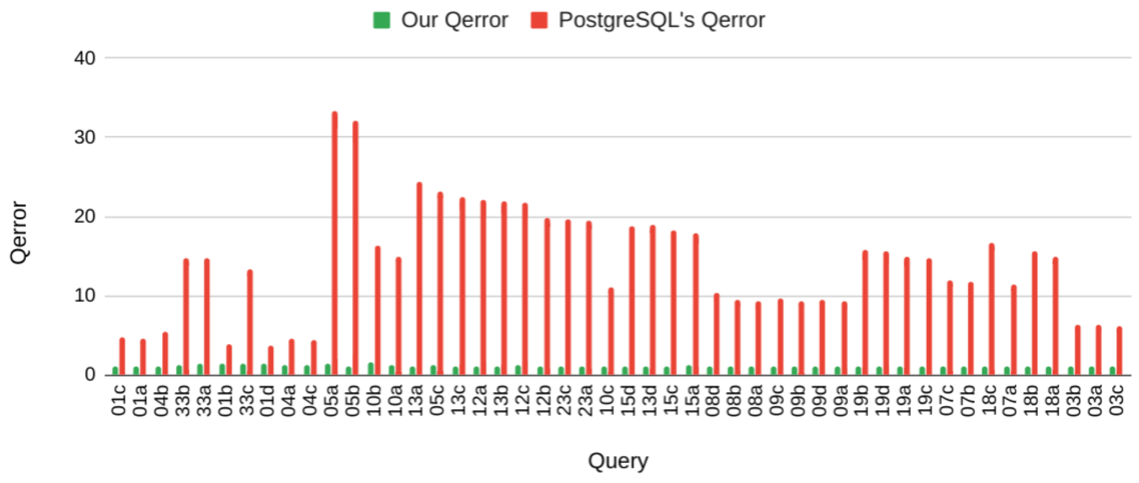


Figure 7.5: Performance on Suite 5

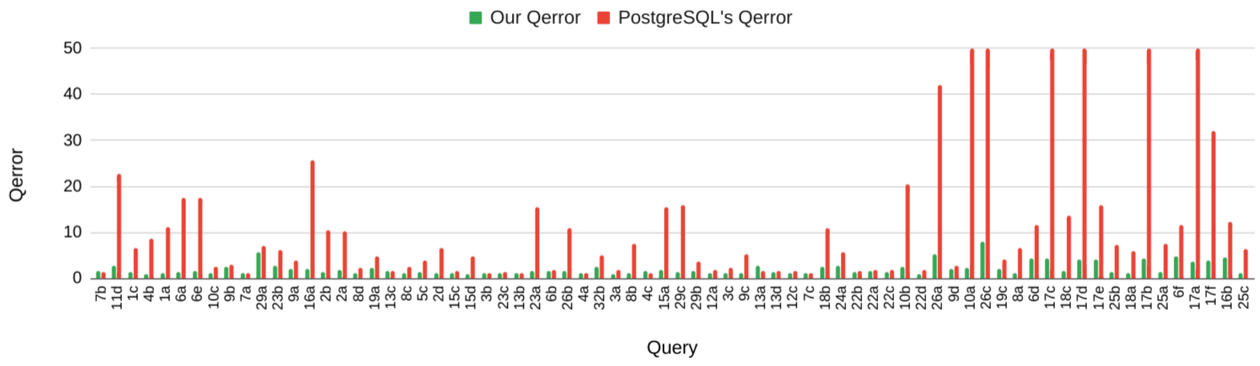


Figure 7.6: Performance on Suite 6

**Suite 5:** The execution times of these 49 plans (Figure 7.5) range from 14s to 2mins. Postgres overestimates the times of these plans due to overestimations on index scans. Here,

$P_Q : [14.1, 14.8, 33.3], O_Q : [1.13, 1.08, 1.5]$. For example, for plan 7a, the actual time was 1.4 mins which we predicted as 1.3 mins but Postgres estimated as 16 mins.

**Suite 6:** The execution times of these 72 plans (Figure 7.6) range from 6s to 27 mins. First 30 plans have times under 1 min. Last 16 plans have times over 16 mins. For some plans, PostgreSQL's Qerror crosses 50 and reaches even 100. But we have kept the bars till 50 for better visibility. Here, $P_Q : [14.3, 6, 121.3], O_Q : [2.1, 1.5, 7.9]$. The large Qerrors in this suite are mostly because of NLJ with index scan operator. Unlike all previous suites where Postgres overestimated times of each plan, here the estimations are mix of underestimations and overestimations. For example, for plan 11d, the actual time of 6 secs was overestimated by Postgres (2.5 mins) and us (18s) whereas for plan 8b, the actual time of 70s was underestimated by Postgres (10s) and overestimated by us (80s).

# Chapter 8

# Spilling

During the execution of a query plan, the executor might have to store the intermediate relations in memory for further operation. However, if the available memory is insufficient, the executor might have to spill data to disk. In our setting, where the possible operators in a plan are scan (sequential and index), sort, join (nested loop join and merge join) and group by with aggregate, following are the operations that can cause spilling:

1. Sort: If enough memory is available, PostgreSQL uses in-memory quicksort. But in case a spilling is anticipated (i.e. when the available memory is less than the incoming input size), PostgreSQL uses external merge sort to sort the input.
2. Index Scan: Due to insufficient available memory, the pages index-scanned previously might be replaced from memory and thus, may have to be scanned again from the disk. This compels us to re-consider our cost model of index scan because in our model, we assumed that any page once accessed from disk can always be found in memory on a repeated access. For now, we do not tackle this scenario and defer its discussion to the future work.
3. Materialisation: In all the other cases, when spilling is anticipated, Postgres materialises the input relation on disk and reads tuples from there. This operation appears as an explicit materialisation operator in the plan.

For the first and third scenarios, Postgres predicts the number of disk reads/writes and multiplies it with a heuristic-based cost of accessing the disk. The CPU costs of all operators remain the same. PostgreSQL's model for spilling is as follows:

1. For External Merge Sort, Postgres uses the standard algorithm which has the following

complexity for number of pages to be read/written from/to the disk:

$$num\_pages = N/B \ * \ \log_{M/B}(N/B)$$

where $N = input\_size, B = page\_size, M = available\_memory\_size$, all in bytes. N/B is the number of runs and M/B is the merge order.

$$disk\_access\_cost = IO\_SEQ\_COST \ * \ 0.75 \ + IO\_RANDOM\_COST \ * \ 0.25$$

2. For materialisation,

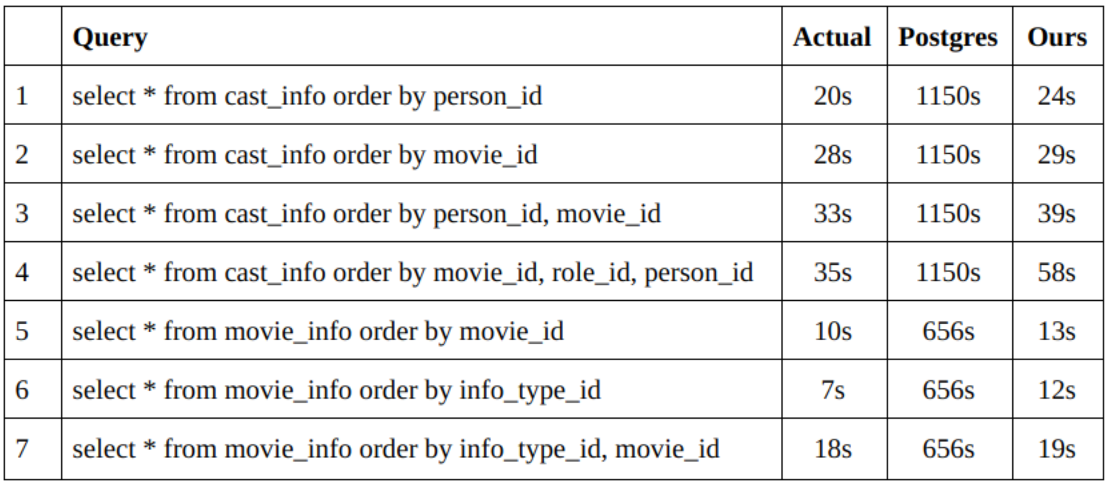
$$num\_pages = N/B$$

$$disk\_access\_cost = IO\_SEQ\_COST$$

Thus,

$$spilling\_cost = disk\_access\_cost * num\_pages$$

In the above model, PostgreSQL was highly overestimating the disk_access_cost because the swap space of disk for storing intermediate relations was actually way more fast. We locally tuned the disk_access_cost and found it to be $IO\_SEQ\_COST \ * \ 0.25$ instead, in both the scenarios.



| | **Query** | **Actual** | **Postgres** | **Ours** |
|---|---|---|---|---|
| 1 | select * from cast_info order by person_id | 20s | 1150s | 24s |
| 2 | select * from cast_info order by movie_id | 28s | 1150s | 29s |
| 3 | select * from cast_info order by person_id, movie_id | 33s | 1150s | 39s |
| 4 | select * from cast_info order by movie_id, role_id, person_id | 35s | 1150s | 58s |
| 5 | select * from movie_info order by movie_id | 10s | 656s | 13s |
| 6 | select * from movie_info order by info_type_id | 7s | 656s | 12s |
| 7 | select * from movie_info order by info_type_id, movie_id | 18s | 656s | 19s |

Figure 8.1: External Sort Examples

Figure 8.1 shows the performance of a few queries on the external sort operator in isolation. As you can see, Postgres estimated same cost of all sort queries on the same relation again, irrespective of the sort attributes. Alongwith our cost model for sort (which accounts for tuple movement cost and differentiates between different ordered set of sorting attributes), the locally tuned disk_access_cost brought down PostgreSQL's cost to predictions that are very close to the actual times.

Figure 8.2 is performance of PostgreSQL's and our cost model on JOB queries. For these queries, we forced the sequential scan operator and let the optimiser pick the join operator. The memory buffers available to Postgres here are almost negligible (a configurable parameter). Although different memory buffer sizes will lead to different spilling scenarios, we show the worst case for maximum possible disk usage. The execution times of the 13 queries in the figure increase gradually from 12s to 3 hours from left to right. Here, $P_Q : [6.2, 4, 19], O_Q : [1.16, 1.1, 1.7]$.

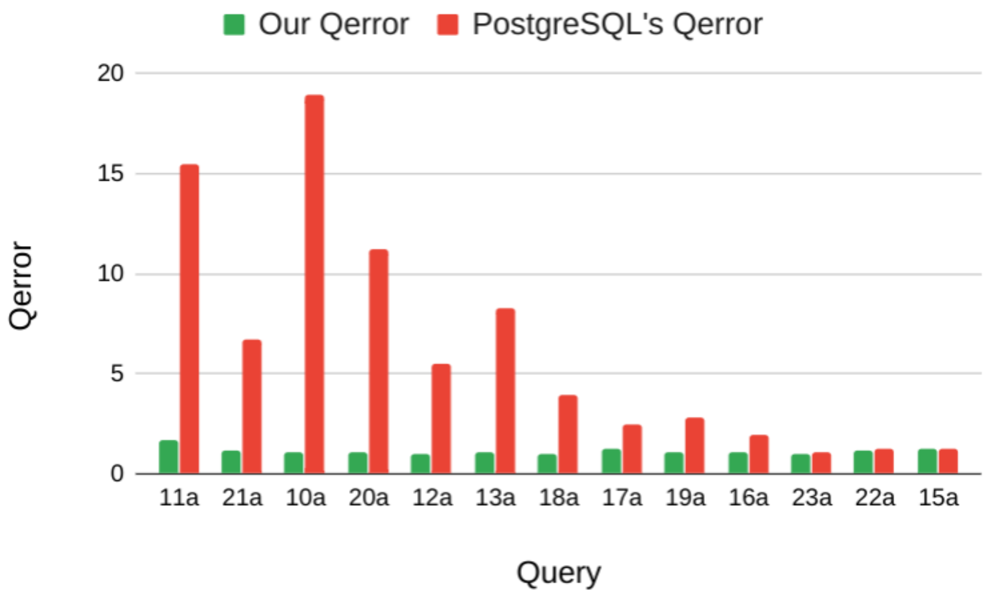


Figure 8.2: Spilling query examples

# Chapter 9

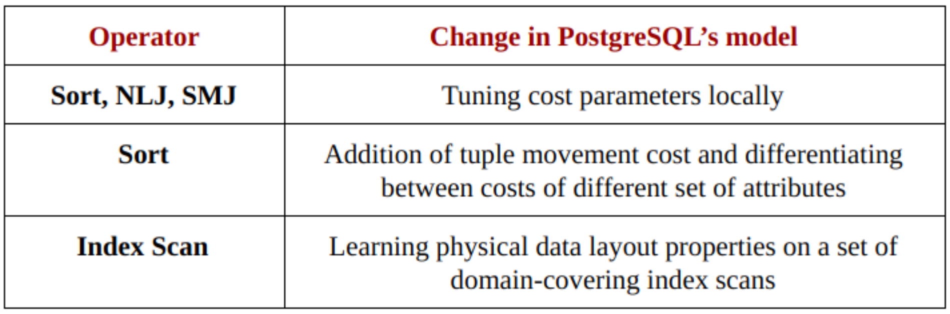
# Related Work

Wu et al. followed up on their tuning [15] approach by trying to tackle the noise in the cost tuning parameters [16], and later ([14]) by using execution feedback on scan operators and mixed cost estimates on internal operators to solve the problem of relative cost modelling (which is useful in problems like query optimisation and index tuning). As part of ML-based research, Chetan et al. [4] used random forest technique on an ensemble of models to give a band of execution time. Ganapathi et al. [3] took plan-level features of a query plan and applied a Kernel Canonical Correlation Analysis approach to predict time and other metrics like CPU usage etc. Akdere et al. [1] went for a hybrid modeling approach (SVM at plan level and linear regression at operator level). Li et al. [7] applied linear regression on all the nodes of a plan followed by runtime scaling. All of these machine learning techniques use either plan-level modelling or same models but at operator-level. They take for instance, cardinality inputs and operator types as features and map execution times on those feature vectors. There have also been deep learning based papers recently ([10] and [12]) that use the neural network approach for each operator and combine the result using another neural network on top. With the advent of big data, some recent papers ([2], [5], [11]) have tried to tackle the problem on big data platforms (like Spark that uses MapReduce algorithm to execute queries in a multi-user environment). Papers like [9] and [13] have tried reinforcement learning as well to find the best plan during query execution, thus bypassing the traditional way of estimating costs and cardinalities.

# Chapter 10

# Conclusion and Future Work

In conclusion, we have tried a different approach of predicting query execution time by minimally changing the cost model of PostgreSQL. We studied PostgreSQL's cost model of each operator and changed the cost functions of each one of them, as summarised in 10.1. For Sort and Join (Nested Loop Join and Sort Merge Join) operators, we locally tuned the cost parameters. For Sort Operator, we identified that PostgreSQL's model does not account for actual movement cost of tuples and does not differentiate between the ordered set of attributes on which sorting is supposed to happen. So we added cost functions to account for these shortcomings. Finally, for index scan, we learned the physical data layout properties based on a set of domain-covering index scan queries by fitting a piecewise linear approximation curve on the execution times of those scans against their cardinalities. We experimentally showed that our approach works well in predicting the cost of query plans accurately by evaluating on more than 350 query plans with 5-6 joins on an average in each plan. Our approach is explainable, easy to debug, generalizable to any query plan and conveniently integrable in the traditional cost models.



| Operator | Change in PostgreSQL's model |
|---|---|
| **Sort, NLJ, SMJ** | Tuning cost parameters locally |
| **Sort** | Addition of tuple movement cost and differentiating between costs of different set of attributes |
| **Index Scan** | Learning physical data layout properties on a set of domain-covering index scans |

Figure 10.1: Summary of our approach

As the future work of the project, we want to predict execution times of the remaining operators (i.e. hash join, index scan on string-valued attributes and in spilling scenario, index-only scan and limit). Further, we want to integrate the model in PostgreSQL to check for any improvement in query optimisation, enhance accuracy of our own model with better approaches (especially for NLJ with index scan), check for any increase in accuracy of deep-learning based cost models by using them only for index scan and, finally, extend the execution environment to multi-user system.

# Bibliography

[1] M. Akdere and U. Çetintemel. Learning-based Query Performance Modeling and Prediction. In *ICDE*, 2012. 1, 36

[2] A. Burdakov, V. Proletarskaya, A. Ploutenko, O. Ermakov and U. Grigorev. Predicting SQL Query Execution Time with a Cost Model for Spark Platform. *IOTBDS*, 2020. 36

[3] 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. 1, 36

[4] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAM*, 2008. 1, 36

[5] A Jindal, L. Viswanathan, K. Karanasos. Query and Resource Optimisations: A Case for Breaking the Wall in Big Data Systems. In arXiv, 2019. 36

[6] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper and T. Neumann. How Good Are Query Optimizers Really? *PVLDB*, 9(3), 2015. 4, 27

[7] 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. 1, 36

[8] L. F. Mackert and G. M. Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *TODS*, 14(3), 1989. 10

[9] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska. Bao: Learning to Steer Query Optimizers. In arXiv, 2020. 1, 36

[10] R. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*, 12(11), 2019. 1, 36

[11] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, W. Le. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In arXiv, 2020. 36

[12] J. Sun and G. Li. An End-to-End Learning-based Cost Estimator. *PVLDB*, 13(3), 2019. 1, 36

[13] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo and J. Antonakakis. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *SIGMOD*, 2019. 1, 3, 36

[14] W. Wu. A Note On Operator-Level Query Execution Cost Modeling. In arXiv, 2020. 1, 3, 36

[15] 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. 1, 3, 6, 36

[16] W. Wu, Xi Wu, H. Hacigümüs and J. F. Naughton. Uncertainty Aware Query Execution Time Prediction. *PVLDB*, 7(14), 2014. 1, 36