On Cost Estimation for the Recursive Relational Algebra

Muideen Lawal

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Muideen LAWAL

Thèse dirigée par **Pierre GENEVES**
et codirigée par **Nabil LAYAIDA**, Chercheur, Université Grenoble Alpes

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Sur l'estimation des coûts pour l'algèbre relationnelle récursive

On Cost Estimation for the Recursive Relational Algebra

Thèse soutenue publiquement le **21 avril 2021**,
devant le jury composé de :

Monsieur PIERRE GENEVES
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur NABIL LAYAIDA
DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES, Co-directeur de thèse
Monsieur FAROUK TOUMANI
PROFESSEUR DES UNIVERSITES, UNIVERSITE CLERMONT AUVERGNE, Rapporteur
Monsieur LADJEL BELLATRECHE
PROFESSEUR, ECOLE SUP DE MECANIQUE ET AEROTECHNIQUE, Rapporteur
Monsieur JERÔME EUZENAT
DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES, Président
Monsieur FEDERICO ULLIANA
MAITRE DE CONFERENCE, UNIVERSITE DE MONTPELLIER, Examinateur

On Cost Estimation for the Recursive Relational Algebra

This work is dedicated to my lovely daughter Fọlàdè, my wife, and to everyone striving to make a difference.

Acknowledgements

First, I would like to express my profound gratitude to my supervisors Pierre Genevès and Nabil Layaïda for their invaluable contribution and support throught the course of my PhD. They gave their time and were involved in the valuable answers to my inquiries during this research work. Without their passionate support, guidance and feedback, this study would not have been possible.

I would also like to thank Cécile Roisin for her encouragement. I appreciate the support of Nils Gesbert. Many thanks to Thomas Calmant, Amela Fejza, Sarah Chlyah, Raouf Kerkouche, Laurent Carcone, and Thibaud Michel for their many productive discussions and advise. Helen Pouchot, thank you for helping with the complex administrative issues during this PhD. I also acknowledge each and every member of the Tyrex team, past and present. It has been such a great honour working alongside every one of you!

I would also like to thank the member of my thesis committee for agreeing to review my work in the first place.

Special thanks to my parents and sibling; Aliu, Rasheeda, Naheema and Toheeb for their unconditional love, prayers and support always. No amount of thank yous will be enough in this lifetime for your contribution to my life.

I would like to acknowledge my wife Olajumoke, I am gratefully indebted to her, for her love,continuous support and understanding during the period of the process of researching and writing this thesis.

Finally, I would like to thank my friends and everyone that has helped me in one way or the other during my study. This achievement would not have been viable without all of you. Thank you all!

You have to do your own growing no matter how tall your grandfather was. — Abraham Lincoln

Abstract

Recursion is becoming a key construct in analytic systems, thanks to the increasing popularity of data structures such as graphs and growth in data over the internet. This resurgence has seen different optimization techniques proposed for recursive queries. Recursive queries are particularly useful for retrieving nodes reachable along deep paths in a graph. Their evaluation involves an iterative application of a function or operation until some condition is satisfied. Cost models remain an essential component of a query optimizer, most important for estimating the cost of query plans and quality plans selection by the optimizer. For recursive queries, however, cost estimation is far from trivial and has received less attention.

One of the challenges encountered in costing a recursive query operator or plan include determining the convergence rate of the recursive. Many systems ignore convergence rate in the data statistics, implementation algorithm and other factors that determine a good cost estimation for recursive query execution. The lack of cost estimation framework support for recursive queries and a validation framework in general for cost model are the main motivation for this work.

In this thesis, we propose a cost estimation technique for recursive terms of the extended relational algebra. This technique uses data statistics and information about the maximum iterative steps needed for recursive evaluation to converge, to estimate the cost of query plans and select an estimated cheapest query plan, in terms of computing resources usage e.g. memory footprint, CPU and I/O, and evaluation time. We also present a cost validation framework where we define a set of metrics and standard specifications for cost model, and the conditions for query plan optimality. These set of metrics and specifications are then used for assessing the efficacy and consistency of plan-selection function of a cost model and they can also serve as a guide for developing advanced cost models. We evaluate the effectiveness of our cost estimation technique on a set of recursive graph queries on both generated and real datasets of significant size. Experiments show that our cost estimation technique improves the performance of recursive query evaluation on popular relational database engines.

Résumé

La récursivité devient un élément clé des systèmes analytiques, grâce à la popularité croissante des structures de données telles que les graphes et à l'augmentation des données sur Internet. Cette résurgence a vu différentes techniques d'optimisation proposées pour cette classe de requêtes. Les requêtes récursives sont particulièrement utiles pour récupérer les nœuds accessibles le long de chemins profonds dans un graphe. Leur évaluation implique une application itérative d'une fonction ou d'une opération jusqu'à ce qu'une condition soit satisfaite. Le modèle de coût reste une composante essentielle d'un optimiseur de requêtes, surtout pour l'estimation du coût des plans de requête et la sélection des plans de qualité par l'optimiseur. Pour les termes récursifs, cependant, l'estimation des coûts est loin d'être triviale et a reçu moins d'attention.

L'une des difficultés rencontrées dans le calcul du coût d'un opérateur ou d'un plan d'interrogation récursif consiste à déterminer le taux de convergence du récursif. De nombreux systèmes ignorent le taux de convergence dans les statistiques de données, l'algorithme de mise en œuvre et d'autres facteurs qui déterminent une bonne estimation du coût de l'exécution d'une requête récursive. L'absence d'un cadre d'estimation des coûts pour les requêtes récursives et d'un cadre de validation en général pour le modèle de coût sont la principale motivation de ce travail.

Dans cette thèse, nous proposons une technique d'estimation des coûts pour les termes récursifs de l'algèbre relationnelle étendue. Cette technique utilise des statistiques de données et des informations sur les étapes itératives maximales nécessaires à la convergence de l'évaluation récursive, pour estimer le coût des plans de requête et sélectionner un plan de requête estimé le moins cher, en termes d'utilisation des ressources informatiques, par exemple l'empreinte mémoire, le CPU et les E/S, et le temps d'évaluation. Nous présentons également un cadre de validation des coûts dans lequel nous définissons un ensemble de mesures et de spécifications standard pour le modèle de coût, et la condition d'optimalité du plan de requête. Cet ensemble de mesures et de spécifications est ensuite utilisé pour évaluer l'efficacité et la cohérence de la fonction de sélection du plan d'un modèle de coût et peut également servir de guide pour l'élaboration de modèles de coût efficaces. Nous évaluons l'efficacité de notre technique d'estimation des coûts sur un ensemble de requêtes de graphes récursives sur des ensembles de données générées et réelles de taille

significative, notamment. Les expériences montrent que notre technique d'estimation des coûts améliore la performance de l'évaluation des requêtes récursives sur les moteurs de bases de données relationnelles les plus populaires.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

INTRODUCTION

Query engines usually generate different equivalent plans during query optimization process among which the best plan is executed. Choosing efficient query plan for execution in query engines can not be effectively accomplished without a properly planned cost estimation techniques accompanied by cardinality estimation. Despite several years of research and techniques on query optimization there is still a huge gap in cost estimation for recursive query plan evaluation.

1.1 Background

Research on database systems are focused mostly on two aspects (i) data storage and access, and (ii) query optimization. Early works on data storage and access database technologies have been concerned mostly with how to efficiently store, retrieve and manage data in data banks. Over the years, the focus shifted on how to effectively speed up query evaluation time and reduce resources consumption paving way for several research works on query optimization.

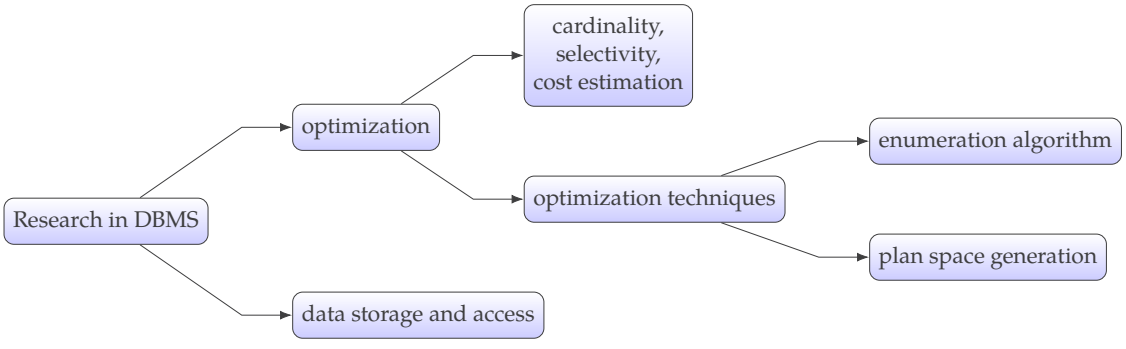


Figure 1.1 – Areas of research in database systems

Figure 1.1 shows the areas of research in database systems.

Since relational model was first introduced by Codd [1], relational database management systems (*RDBMS*) has been by far the most popular and dominant data analytic systems for many decades now and they are effective in storing, managing and retrieving data and are *ACID*-compliant[1]. The relational model stores data in a predefined schema and is implemented through the standard Structured Query Language (*SQL*) interface in many relational database management systems (*RDBMS*) like MSSQL Server, Postgres, Oracle, MySQL MariaDB, etc.

Over the years, there has been a paradigm shift from centralized *RDBMS* to distributed big data platforms, this is in part due to the need for large scale query processing on data from heterogeneous sources and increasingly growing size of data. Distributed analytic engines like Hadoop [2], Spark [3] and Flink [4] among others, provide means of handling computation on massive amount of data by distributing data and computation across several machines thus allowing them to efficiently handle computation on very large datasets. Many data-intensive applications have since taken advantages of these frameworks.

To efficiently store and extract values from linked data, researchers started focusing their attention to graph databases. Graph data are usually represented by nodes and edges where any two entities (nodes) are possibly connected by an edge. Traditional databases can store graph data but unfortunately, they were not designed for that purpose; they are good for storing and managing structured tabular data. If many relationships exist between data, *RDBMS* usually requires several joins in order to evaluate a given query [5], hence, they are not as efficient in processing graph data. Relationships among data are directly represented in a graph database, allowing queries to directly use the graph structure [6]. Graph database are generally faster compared to traditional database when it comes to finding complex relationships between the nodes.

Data stored using *RDBMS*, graph databases or big data systems needs to be retrieved, transformed, and updated from time to time. For *RDBMS*, SQL has been the de-facto query language for many traditional database systems and there are different SQL versions adopted by different vendors. As for querying graph data, most systems have their own domain-specific language e.g. SPARQL is the standard language for querying RDF data, Cypher for Neo4j and AQL for ArangoDB, and some even offered an SQL flavoured dialect to add some familiarity for traditional database users migrating to their platform. The main idea behind writing query in high-level languages like SQL is to allow users to write queries without having to bother about the implementation or execution details of how the query will run. Unfortunately, user-written queries may not always be optimized in such a way that query execution will efficiently utilize and/or exploit the opportunities or computing power provided by the analytic systems. For data application to efficiently

[1] they are characterized by the Atomicity, Consistency, Isolation and Durability features where data integrity is enforced in database transactions regardless of errors, power failures, etc.

benefit from the opportunities offered by both *RDBMS* and distributed big data platforms, there is a need for query analysis and optimization.

Query optimization is a process of generating different alternative plans or ways of executing a given query and selecting the best query execution plan for evaluation [7, 8] from those set of alternatives. It is a challenging task that has attracted several interests in the research community over the last few decades resulting in different techniques and implementations [9–16] in various commercial and open source database systems. Some of these works focused on improving plan space generation which is a process that involves generating alternative plans and algorithms for implementing underlying operators efficiently.

Advances in this areas (query optimization) led to further questions on how to quantify what determines a better query plan. To answer this question, early researchers of database query optimization devise some set of predefined rules known as heuristics for determining what order and combination of query sub-expressions forms a better execution strategy. The reliance of heuristics is not entirely a bad idea but many of these transformation rules are not always the best for performance. The query optimizer therefore needs additional information to make better decisions regarding the ordering or arrangement of expressions on query trees of candidate query plans to be considered and eventually selected for execution. Information about the data properties (i.e. data statistics) e.g relation and column cardinalities, index selectivity, etc. are crucial to understanding how much an operator or expression costs and the estimate of the amount of time the query engine will spent evaluating them. The process of estimating the cost for each operator and query plan is known as cost estimation.

Query evaluation is composed of two key components; the query optimizer and the query execution engine [10]. In a typical query optimizer, queries are represented as a tree of operators know as the query execution plan *QEP*. A traditional query engine is made up of three main components shown in Figure 1.2, (i) search space consisting of execution plans, (ii) cost model to estimate the cost of each *QEP*and assign the individual cost to the *QEPs* and cardinality estimation to keep track of the changes in the size throughout the optimization process and (iii) enumeration algorithm and plan selection, to determine the choice of algorithm to be used or implemented for the execution of a particular operation. Equivalent *QEPs* are generated per query by the optimizer in the *search space*, each with different evaluation *cost*.

The query engine includes a query optimizer, a crucial component in charge of searching for equivalent plans but in which operators are rearranged for efficiency search purposes while preserving the semantics of the initial query. The query optimizer requires a cost estimation technique that selects a best plan, i.e. that provides a priori a better evaluation time and minimizes resources usage. For a given query sent to a query engine for evaluation, the optimizer first translates the query into a QEP, then generates a potentially huge number of equivalent QEPs.

Recursion expresses a category of query that has several important applications in

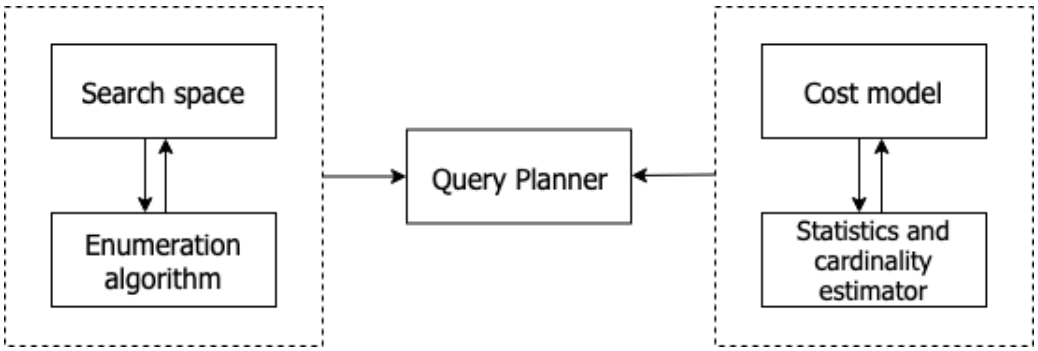


Figure 1.2 – Query optimizer architecture

data analytics especially with the volume and veracity of data in the era of *big data* and they are very useful in exploring and finding relationships among connected data. Typical recursive algorithms used in data-instensive applications include *Page Rank, Connected Component, Shortest Path etc.* Recursive evaluation involves an iterative application of a function or operation until certain conditions (of termination) are satisfied –know as the fixpoint. The growing need for recursive query support and processing in analytic systems has been driven partly by advances in large-scale data analytics and knowledge database development, popularity of data structure such as graph, growth in semantic web and, the rapid growth of data over the internet [17–20].

A number of studies have been conducted on recursive queries optimization including [17, 21–23] and more recently [4, 24–26]. Recursive operators are often difficult to implement and it is even more challenging to optimize [27] their execution. In order to efficiently handle recursive query evaluation, some of the techniques that have been proposed include implementing recursive operator in an intermediate language that compiles to standard SQL for evaluation on *RDBMS* [18, 19, 26], and others include leveraging the computational power *i.e.* task execution in batch over clusters of machine offered by distributed data analytics framework like Spark and Hadoop e.g. *REX* [28], *BigDatalog* [25], *Haloop* [29], [24] and *Flink* [4] which natively support iterative workflows. Whichever method or platform chosen to implement the fixpoint operator, optimization process crucially relies on *QEPs* enumeration. *QEP* enumeration is carried out using the optimizer's cost estimation combined with cardinality estimation techniques to form an overal metric that will then be used for comparing equivalent *QEPs*. All equivalent *QEPs* generated for a given recursive query produce the same output, *i.e. number of results* but they differ in terms of cost and amount of time that will be spent evaluating them.

Despite the recent attention received by recursive query evaluation, optimizers for recursive operators still suffer from inadequacy of selecting best *QEP* for execution. Iterative algorithms execution incurs significant amount of overhead [28], one reason is that recursion is a repeated operation performed until no new results are available for further computation or certain termination criteria is satisfied.

1.2 Motivation

While recursive queries can appear in different domains and their applications are diverse, we take a look at an example from the ongoing Covid-19 pandemic. To give more context, a single host of Covid-19 virus can propagate the disease to several hundreds or thousands of people. Assuming we have a graph, where the nodes represents people in a locality or country and edges are the interaction between those people. In Example 1.1, we examine a query which tracks the spread of Covide-19 infection.

Example 1.1 ***Covid-19***

During the ***Covid-19*** *pandemic one of the common task is identifying the spread of coronavirus. The question that is often asked for any patient is who have you been in contact with and who has been in with those people as well. This question can easily be formulated as a recursive query.*
Let us assume that we have a query to retrieve a list of person ?x that JohnDoe *(a patient) was in contact with and the people that have been in contact with these people.*

$$\left.\begin{array}{rcl} ?x & \texttt{wasWith+} & ?y \\ \texttt{JohnDoe} & \texttt{inContactWith} & ?x \end{array}\right\} Q_1$$

In Example 1.1, line 1 *wasWith+* represents the transitive closure of *wasWith*.

There are several possible set of solutions or ways to evaluate the query in Example 1.1. The possible set of solutions (loosely translated into *MuRA* [26]) are as follows;

1. `P1`: Filter(John, Join(inContactWith, Fixpoint X $\rightarrow$ wasWith $\cup$ X/wasWith))
2. `P2`: Filter(John, Fixpoint X $\rightarrow$ Join(inContactWith, wasWith)) $\cup$ X/wasWith)
3. `P3`: Fixpoint X $\rightarrow$ Join(Filter(John, inContactWith), wasWith) $\cup$ X/wasWith

It should be noted that there are other ways to evaluate Q_1 above.

Similar to treating the query in Ex. 1.1 first, as two separate queries, and joining their results, plan 1 (`P1`) starts by computing the transitive closure of everyone that have been with (`wasWith`) someone and joining that with a list of `?x` that `JohnDoe` has been in contact with (`inContactWith`). A downside of this approach is that we perfom a join as many times as the number of `wasWith` which might be quite expensive since only the connections to `JohnDoe` are only needed in the end. `P1` runs in $0(n^2)$ time.

Another option in `P2` is to push the join inside the fixpoint. This means we start from `inContactWith/wasWith` and iteratively add `wasWith`, then filter by `JohnDoe`. This is a fairly better approach than `P1` since the number of joins performed is reduced.

The last approach we consider is `P3` where the filter is pushed inside the join in the fixpoint operator. This approach is very common in query optimizer and the idea is to filter early so as to reduce the intermediate result size or the degree of nodes to be

computed (inside the fixpoint in this case). It is a lot more efficient approach than *P1 and P2* and runs in $0(n)$ time i.e. the number of solutions is linear to the number of nodes in the graph.

In the end, Q_1 has different different equivalent plans with varying cost of execution. In order for the optimizer to make this clever decision of selecting the cheapest query plan *e.g. P3* in this example, a better cost model that effectively the best cost estimate for each *QEP* is needed.

Cost estimation plays a bigger role in query optimization, faciliting the selection of cheapest query execution plan by the query planner —cheapest in terms of computing resources usage *e.g. memory footprint, CPU and I/O* and evaluation time. The classical approach to cost estimation is given in [7] and have since served as the foundation for cost model implemented in mainstream database systems. In this approach of [7], the cost model is built for the SPJ (Select Join Project) families of query and the cost is estimated in terms of the CPU and IO cost needed for a query to be processed.

A cost model takes as input *QEPs*, data statistics collected on the base relations (e.g cardinality, number of distinct tuple per column) and other metrics like *selectivity estimate*, *access path* etc. The effectiveness of *cost-based optimization* for non-recursive constructs has been studied in [7, 10, 20, 30–32], however, for recursive terms, cost estimation has received less attention and as a result existing systems lack a better cost estimation technique for recursive query evaluation.

QEP cost estimation is necessary to ensure that the cheapest query plan selected by the optimizer intelligently selects operators precedence (as with `P3` above). Tuples or rows that will not be part of the final result should be discarded as early as possible. For example, filter operator reduce the intermediate result size of data by eliminating tuples that does not satisfy a given predicate. Evaluating this operator as early as possible in the query tree will improve the query performance.

Recursion usually involves iterating over a set of elements in a relation in a certain number of steps, say $\mathbb{N}_j$, where j is the depth of the recursive tree. Estimating $\mathbb{N}_j$ has proved to be a great challenge in cost estimation for this class of queries. We have observed that existing systems in this domain most often assume a constant number of iteration as in [33, 34]. One downside of this approach is that a query could take say 50 steps for the recursion to converge on a dataset and even more steps if the number of nodes are increased. Another pitfall observed is that *P3* will take less steps than *P1* in Example 1.1 above since a filter is pushed earlier inside the fixpoint reducing the number of nodes to be computed. As a result, assuming a constant number of steps will eventually lead to a bad estimation of the cost. Due to lack of specialized recursive operator in many database systems, there has been limited interest in cost estimation techniques that focuses on recursive query evaluation.

Many work have been done to improve the cost estimation accuracy, but as we will show later, these techniques are not adapted for every situation. The lack of experimental validation to demonstrate the effectiveness of recursive cost estimation proposed in [23]

for example or an oversimplified cost estimation in [21] are examples of the limitations of existing works for recursive cost estimation.

Accuracy of cardinality estimation is also important as this can greatly influence the effectiveness of cost model and subsequently, the *QEP* quality [35]. Some works on cardinality estimation [35–41], have demonstrated the impact of cardinality estimation on a query optimizer. The cardinality of the base relation of a given dataset for a given query is obtained either from the table of the database catalogue or just before query evaluation time. The cardinality retrieved is part of the statistics used by the cost model and this statistics are usually propagated through the entire query tree. The intermediate result size which is the cardinality of each operator on the query tree may change (increase or decrease) depending on the operation performed and the cost model uses this information to estimate the cost of each operator in the *QEP*, allowing the optimizer to chose between different *QEP* alternatives based on their cost. *Selectivity* refers to the set of tuples in a relation that satisfy a given predicate [7, 30]. Selectivity factor is an important property for estimating the result size of operators that involves the elimination of tuples from a relation such as *join*, *filter*.

An accurate cardinality estimation does not automatically translate into a significant improvement in query evaluation time since cost estimation in a query optimizer is a complex combination of factors. It is therefore, important to have an accurate cardinality estimation together with an efficient cost estimation technique. We believe that in order to achieve a "near-optimal" estimation of *QEPs*' cost and accurate cost model, cost estimation in query optimizers should be considered with equal importance as cardinality estimation. Finding a good balance between both (cost and cardinality estimations) will ultimately ensure accurate *QEPs*' cost estimation and subsequently improve query runtime performance.

Another aspect that is lacking in query optimizers is a proper cost validation framework. Many works on cost models in query optimizers rely on the query evaluation time for comparing how effective a cost model is compared to others. This information alone does not give the needed insights into the behaviour of cost model. Modification of cost model input paramters introduce changes to the behaviour of the cost model and eventually the quality of the plan selected. The lack of standard evaluation approaches makes it complex to track these changes. In practise, the cheapest plan selected by the optimizers' cost model are not always the plan with the minimum evaluation time and as a result, errors are introduced in the estimation. And to which extent do we consider this errors acceptable or query plans near-optimal and how do we set the bounds? We provide answers to these questions in this thesis.

As mentioned above, our main focus are on recursive queries. A recursive query contains a fixpoint operator with other non-recursive constructs to form a query tree. This work focuses on generating efficient query plans, one that has the minimum cost among a set of alternative plans.

1.3 Contribution

To address the limitations of existing systems, we present a cost model for recursive query optimization which allow for efficient selection of "cheapest" *QEP* for evaluation. Based on the aforementioned, we summarize our contribution on cost estimation for recursive query evaluation as follows;

1. **Maximum iterative steps in a fixpoint operator.**
   The first problem we address is determining the number of iterative steps needed for a recursive query evaluation to converge. We present a technique for estimating this number and gather the results during a fixpoint operation in Chapter 4. The steps estimation is important for the cardinality estimation and determining the estimated cost for the recursive evaluation.

2. **Cardinality estimation.**
   Accurate cardinality estimation plays an important role in the quality of query plans selected by an optimizers' cost model since its one of the statistical information the cost model rely on. Although cardinality estimation for non-recursive constructs has been well-studied, for recursive queries this is not the case. We present a cardinality estimation technique in Chapter 4 for fixpoint operator and extend the work to rdf graph summaries by Stefano et al. [36] to accurately estimate the cardinality of recursive operator. In particular, this allows us to extend the definition of the relational algebra presented in [26] with cardinality information of each operator.

3. **Cost estimation.**
   In Chapter 4, we present a cost estimation technique that utilizes *(1) and (2)* above and other relevant set of parameters or statistics to efficiently estimate the cost of query plans. We present the cost specification, assumptions and functions for each relational operator. Specifically, we present a novel approach for the cost estimation for the fixpoint (recursive) operator. We walk through the computation steps involved in the fixpoint operator, giving a deep insight into how the cardinality and cost is computed at each step.

4. **Validation framework.**
   We introduce a cost validation framework in Chapter 5. We begin by defining a set of metrics and standard specifications that cost models are required to conform to. We also define the conditions for optimality of query plans selected by an optimizers' cost model. By using a rank-based approach and an error model, this framework does not only provide a way for assessing the effectiveness of a cost model's plan-picking function but also a means for comparing different query optimizer's cost functions.

5. **Implementation and evaluation.**
   Finally, our cost estimation technique is implemented based on the algebra presented in [26]. Using both real world and generated graph datasets and regular path queries, we conducted a comprehensive performance study, comparing our estimation technique with existing systems. Results show that regardless of the size of the graph or complexity of queries, our technique performs better on average compared to existing systems.

This work enables query engines that support recursive query evaluation to effectively select the cheapest query plans hence, an improved query evaluation time and resources utilization. This work also serves as a foundation for the understanding of recusive query plans and cost estimation and highlights the difficulty of recursive query cost estimation.

All experiments carried out in this work were conducted on recursive terms and particularly tested for the extended relation algebra presented in [26]. We show that our approach can be easily implemented in any mainstream database or analytic system that support fixpoint operator or recursive query evaluation.

1.4 Thesis Outline

This thesis is structured as follows; in Chapter 2, we present the definition of terms, the data model and notations used throughout this thesis. We also discuss query optimization techniques, heuristics, and cost-based query optimization. In addition, we discuss recursion and recursive query evaluation which the contributions in this thesis are centered around. We give and overview of transitive closure algorithms and we conclude the chapter by presenting the algebraic framework which will be referenced throughout this work and for which our cost estimation techniques have been designed on. A survey of cost estimation methods are presented in Chapter 3, we categorize cost models and examine the different cardinality and cost estimation approaches in existing systems for both non-recursive and recursive queries both in centralized and distributed settings. From the lessons learned and inspiration derived from the state-of-art, we set a baseline for formulation of our cost estimation techniques.

In Chapter 4 we present our novel cost estimation technique for fixpoint operator based on the extended relational algebraic framework of Chapter 2. We also present a cardinality estimation technique for fixpoint operator based on graph summarization technique (SumRDF) [36]. Cardinality and selectivity estimation together with data statistics are used as input for the cost model to efficiently estimate the cost for recursive *QEPs*.

We present a cost model validation framework in Chapter 5, outlining the criteria for determining the optimality of query plans and defining the method to do so. In Chapter 6, we described the implementation for the cost formulas proposed in Chapter 5. The underlying architecture, design choices are also explained. We experimentally

evaluate the implementation presented and we demonstrate its effectiveness. Finally, we give the conclusion and directions for future work in Chapter 7.

CHAPTER 2

PRELIMINARIES

To facilitate readers' understanding of the work presented in this thesis, it is important to introduce the notations and provide the foundation and necessary technical background used throughout this work; this we present in this chapter. Relational and graph model, query optimization, and recursion are some of the concept discussed herewith.

2.1 Introduction

The standard SQL does not support recursive queries until SQL99, however, some vendors like Oracle and IBM already supported this class of queries. Even though many *RDBMS* systems now support recursive queries in the form of SQL "RECURSIVE CTE" or similar dialects, this support is still very limited in that they can mostly handle the case of linear recursion in which reference to recursive or moving part of the query is restricted to one call. Recursive patterns in graph are very common. In order to mine answers from graphs, recursion can be nested and merge as (in [26]) at depths depending on the size and number of connections between nodes of the graph.

In this chapter, we discuss relational and graph data models and give some important definitions. In Section 2.2, we present the relational model, storage methods, query language and the relational algebra. In addition, we introduce the graph model in Section 2.3 where we give examples of graphs, we describe the storage facilities for graphs and also the different languages for querying them. We also introduce query optimization in Section 2.4 and describe the different optimization approaches employed in query engines. Finally, in Section 2.5, we introduce recursion and present transitive closure algorithms and fixpoints.

2.2 Relational Model

The relational model was first introduced by *E. F. Codd* in 1970 and have since been the foundation of relational dabatase management systems. The simplicity of the relational model helped it gain wide acceptance in the research community and data analytics sphere. The relational model is based on the concept of mathematical relation and relations are manipulated via the relational algebra and query languages. The relational model uses attributes and the relationships between those attributes.
We give some definitions to describe the relational data model as follows;

Definition 2.2.1. ***(Relation)*** *Let* R *is a relation over domain values* $A_1, A_2, \ldots A_n$ *then,*

$$r(R) \subseteq dom(A_1) \times dom(A_2) \times \ldots dom(A_n)$$

where $r(R)$ *is an instance of the relation* R *and* $\{t_1, t_2, \ldots t_n\} \in r(R)$ *are the tuples*

Definition 2.2.2. ***(Attribute)*** *An attribute defines the properties of that relation.*

Definition 2.2.3. ***(Tuple)*** *A tuple refers to a row of a relation.*

Definition 2.2.4. ***(Cardinality)*** *The cardinality of a relation* R *refers to the total number of rows or tuples contained in a relation* R *table. The cardinality is written as* $|R|$ *or* $\mathtt{rowCount}(R)$.

Table 2.1 – Relation example

id	Employee	Salary
1	Ola	8
2	Peter	7
3	Jumoke	5
4	Abiola	3

In Table 2.1, each row represents a tuple; a single item of the relation. Each cell or field represents an attribute and the columns consist of a set of attributes; labeled items of a tuple.

2.2.1 Storage

RDBMS is used to store data in a relational model. Basically data is organized or stored in a database table as rows and columns where the rows hold the information and the

columns refers to the data type and category. *RDBMS* are generally *ACID* (Atomicity, Consistency, Isolation, Durability) compliant.

2.2.2 Query Language

Data is updated and retrieved in relational database systems using the Structured Query Language *SQL*, the standardized query language for the relational model. Many systems including commercial databases and big data platforms have adopted the *SQL* standard while some have only implemented a flavoured version of it. At the minimum, a query language allows users to write clear and simple queries without having to understand about how the query will be processed [42].

> **Definition 2.2.5.** ***(Query)*** *Given a database instance* D*, a query* q *involves mapping* D *to a relation* $\mathsf{q}(\mathsf{D})$*.*

```
01 |  SELECT *
02 |  FROM Emp
03 |  WHERE age = 25;
```

Listing 2.1 – Query language example

Listing 2.1 shows an example of an *SQL* query that selects all employees that are 25 years old.
Query evaluation is done in several stages as follows;

- parsing/transformation: the stage includes syntax validation, attribute and access permission and query transformation into algebraic expression
- optimization and code generation: the optimization phase involves rewriting algebraic expressions into more efficient ones leading to the construction of several alternative physical plans. Based on the statistics of the data and the execution environment, a cost model is used to select the cheapest *QEP*.
- query evaluation: the cheapest *QEPs* selected by the help of a cost model in previous step are then executed.

2.2.3 Relational Algebra

In this section, we discuss how the database uses the query language to retrieve and update stored data, the relational algebra (*RA*) allows for describing operations on relations using formal mathematical notation.

Definition 2.2.6. *A relation algebra consists of a relation* R *together with a set of operations to be performed on* R.

A relational algebra operator takes one or more relations as input which can be base relations (read directly from the database table) or outputs of intermediate relation(s). An operator could be unary, binary or a closure. Unary operators accept only one relation, binary operators involve two relations as input while closure operators accept relation as input and output one or more relations.

Some examples of *RA* operators include;

- Filter: an example of unary operator which takes a selection predicate f and a relation R. *Filter* involves selecting only tuples that satisfy a given predicate condition. The relation returned after a *filter* operation has the same schema as the initial relation but whose tuples are a subset of this relation. *Filter* is written as $\sigma_f(R)$.
- Projection: written as π, a projection operation takes a relation as input and returns a subset of the initial relation. Usually, the output relation have the same number of tuples as the input relation unless duplicate attributes are removed.
- Join: Join is a binary relation written as $\bowtie$,.. Given a set of terms R and S, this operator involves combining compatible pairs of tuples from R and S. The two relations must be $\mathtt{join}$ compatible. The $\mathtt{join}$ $(\bowtie)$ of two terns R and S is written as $R \bowtie S$.
- Union: Given two terms R and S, the $\mathtt{union}$ $(\cup)$ operation corresponds to the compatible set of tuples from R and S. The result of a union operation is a relation $R + S$. The $\mathtt{union}$ of two terms R and S is written as $R \cup S$.

A comprehensive description of these operators used in this work will be given in subsequent chapters.

2.3 Graph Model

Some of the inherent limitations of *RDBMS* which has been designed for general tables has led to the development of graph databases, optimized for storing and querying graphs. Essentially, a graph is a collection of nodes and edges. In the graph model, entities are represented as nodes and their relationships as edges.

Definition 2.3.1. ***(Graph)*** *Let graph* G *be an undirected graph containing a set of vertices* V *and edges* E *respectively, we define* $G = (V, E)$ *such that* $E \subseteq V \times V$*, where* V *is a set of nodes and* E *is a set of edges. Assuming* $E(s, t)$ *is a relation, there exist an edge* y *that is mutually reachable from* x *iff* (x, y) *is a tuple of* E.

The basic entity of a graph is the node, connection between two nodes is established through an edge.

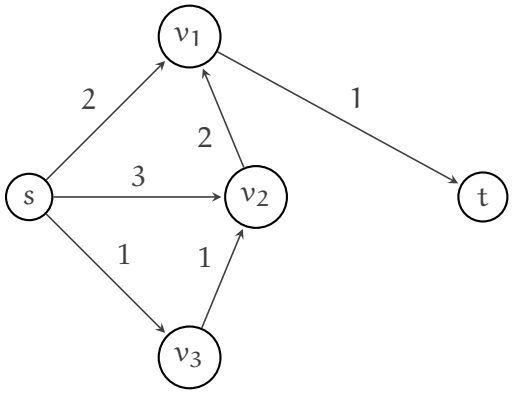


Figure 2.1 – Graph example

Definition 2.3.2. ***(Path and path length)*** *Let* $G = (V, E)$*. A path* p *is a sequence of nodes* $v_1, v_2, v_3, ...v_k$*,* $v_i \in V$ *such that* $(v_{i-1}, v_i) \in E$ *for* $i = 1, 2, 3, ...k$*. The length of the path is the number of edges in the path.*

Definition 2.3.3. ***Graph size*** *Given a graph* $G|G = (V, E)$*. The size of graph* G *is the defined as the sum of the total number of nodes* $|V|$ *and edges* $|E|$ *in the graph.*

$$|G| = |V| + |E|$$

Definition 2.3.4. ***(Transitive Closure)*** *Let* $G = (V, E)$ *with* $v, w \in V$*. The transitive closure is the graph* $G^+ = (V, E^+)$ *with* $E^+ = (v, w)|v \rightsquigarrow w \text{ in} G$*.*

Definition 2.3.5. ***(Fixpoint)*** *A fixpoint of a function* $f \in D \to D$ *is an element* $x \in D$ *such that* $f(x) = x$*.*

2.3.1 Storage

The last decade has seen an emergence of many different graph databases. Research on graph data management has mainly been focused on two aspects, *(i)* the native data stores, *(ii)* non-native data stores.

- Native data stores: the native data storage system are usually RDF model compliant which could be in-memory or disk-based [43]

- Non-native data stores: this category of graph storage system make use of relational systems *e.g.* Apache Jena, C-Store [16], Neo4j, SQLGraph [18], MuIR [26] or *NoSQL* systems *e.g.* OrientDB or *XML/HTML/Web API*-based *e.g.* RSS feeds.

> **Definition 2.3.6.** ***(RDF)*** *Resource Description Framework (RDF) [44] is a* W3C *recommendation for representing web resources or knowledge bases. It describes the syntax and encoding for writing, exchanging and reuse of structured knowledge.*

Entities in RDF are uniquely identified by their IRIs but they can also be literals and blank nodes. RDF is built on the *subject-predicate-object (s, p, o)* triple pattern model where the *predicate* indicates the relationship between the *subject* and the *object*

2.3.2 Query Language

The graph data model allow paths to be specified in queries by the user [45]. There is no particular standard query language, each systems tends to have their specific language for retireving and updating records stored in graph databases. SPARQL is the standard query language for RDF graphs. AQL (Arango Query Language), a declarative language for querying record stored in ArangoDB, OrientDB uses a flavoured SQL dialect and Neo4j uses CQL (Cypher Query Language). Regular Path Queries (RPQs) [46], Conjuction and Union of Regular Path Queries (UCRPQs) [27, 45, 47] are also query languages for graph databases, navigational in nature as they are used to query labeled edges or paths in the graphs.

> **Definition 2.3.7.** ***(SPARQL)*** *is the query language for* RDF

```
01 | PREFIX sd: <http://example.org/knowledge-base#>
02 | PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
03 | SELECT *
04 | WHERE {
05 |   ?person sd:livesIn ?city
06 | }
```

Listing 2.2 – SPARQL query example

> **Definition 2.3.8.** *Regular Path Queries (RPQ) are regular expressions that are matched against labeled directed paths in graph databases. A regular expression* R *of an RPQ over labeled edges is of the form;*
>
> $$\mathrm{RPQ}(x, y) := (x, R, y)$$

where x and y are the edges

Examples of an RPQ query is given in Listing 2.3. RPQs permit users to formulate queries about arbitrarily long paths in graphs [48] and they can contain more complex paths, a complete description can be found in the work of [27, 45–47].

```
01 |  ?person livesIn/locatedIn ?city
```

Listing 2.3 – RPQ query example

2.4 Query Optimization

For a given query, there could be several ways to execute the query and the optimizer generates different equivalent *QEPs* accordingly among which a best plan is selected for execution. The difference in equivalent query plan can be observed in terms of their evaluation time and computing resources consumption. Query optimization process is basically carried out in a sequence of steps where a given query is first subjected to a series of transformations then all possible combinations of operators are generated making up different execution plans. The query engine then select the cheapest *QEP* that produce an output equivalent to the original query. We discuss query processing strategies in the next section.

2.4.1 Bottom-up vs Top-down

There are two prominent architectures that has been adopted for query optimization during the last few decades (1) System R, bottom-up dynamic programming optimizers [7, 49] and (2) Volcano top-down memoization optimizers [50, 51].

Bottom-up style enumerates plan in a bottom-up fashion using dynamic programming [7, 49]. It was first prototyped at IBM Research and has been implemented in various database systems such as IBM DB2 and Oracle. In terms of search space and plan generation, the original query is transformed into semantically equivalent algebraic expressions otherwise known as the logical plans. This stage is referred to as the query rewriting stage. Semantically equivalent expressions or logical plans are translated into a tree of physical operators –the physical plan, where the search space is further broadened or enlarged and an efficient *QEP* is selected for evaluation with the help of a cost model. System R consider properties of a query like columns ordering which are then used in selecting the efficient plan for execution. Optimization process in bottom-up approach is two-fold; rule-based query rewriting followed by the application of cost estimation. Volcano style on the other hand translates queries into logical expressions and apply transformation rules that maps algebraic expressions to other expressions. Physical plans then implements the logical expressions and the cheapest physical *QEP* is sent for execution. This process is exhaustive and all possible transformation rules are applied in a top-down manner. In contrast to bottom-up style, Volcano style optimizers are single-phased where all algebraic

transformation are cost-based and the mapping of algebraic expression to physical ones is done in a single step. Systems that implement the Volcano style optimization include Apache Calcite (and by extension Flink), SQL Server, etc.

Plan enumeration in bottom-up optimizers uses dynamic programming where only the cheapest plans for sub-expressions are considered and the final search tree consists of the possible solutions of orders in which these expressions or plans can be combined to retrieve the query results. This is achieved in an incremental way such that enumeration starts from lowest level nodes in the access plan then proceed by enumerating combination of two of these type of plans up until the highest level. There are $n!$ permutation of ways in which access plans can be combined for a query but dynamic programming reduces this from $O(n!)$ to $O(n^{2^{n-1}})$.

Enumeration of plans in top-down optimizers are done through backward chaining using branch-and-bound search and memoization [50, 51] this avoids the duplication of effort. The optimizer checks the logical and physical properties of expressions to determine whether the query expression has already been optimized, in which case it does not re-apply this optimization. If the optimization has not been performed before, it applies transformation and implementation rules and modify the properties with an enforcer. A bound derived from siblings and parent expressions is used as baseline in pruning the search space for expressions that do not have to be enumerated.

Cost estimation for both query optimization approaches are similar. Cost is estimated for each operator or sub-plan and the cost for each intermediate operators or sub-plans are combined into a final estimated cost for each *QEP* allowing the optimizer to select the cheapest query plan for execution. The cost model is a combination of arithmetic formulas that rely on the CPU and I/O cost. The cost model takes as input, the data and index statistics e.g cardinality, column distinct values, intermediate result size, etc., the predicate in the query, access path and particular order of evaluation of the query [7]. In both query optimization architectures, the cost estimation is performed in a bottom-up manner since the final cost of each plan is the cumulative sum of that of all of its sub-plans.

2.4.2 Heuristics vs. Cost-Based Query Optimization

Choosing an efficient *QEP* from several possible alternatives is a very hard task that the query optimizer must perform. There are two common approaches that the optimizer uses to carry out this task. Heuristics is the first approach which is based on applying equivalence rules to systematically generate equivalent expressions to a given expression without the knowledge of statistics of the dataset [11]. They are usually applied to deal with huge search spaces and they reduce the number of choices.

On the other hand, cost-based optimization involves the enumeration of all possible plans in the search space and it assigns a cost to each plan in the search space. The cost is a relative measure of the amount of resources a particular *QEP* will use during the execution. The final cost of a *QEP* is the summation of the cost of all its sub-plan or operations. The

query optimizer choose the *QEP* with the cheapest cost after enumeration for execution.

One might think that heuristics are sufficient for query optimization given the overhead sometimes incurred by cost estimation, this might be true for trivial queries only. Recursive queries are a complex class of queries. In particular, if they contain a *fixpoint*, therefore, their evaluation requires careful planning and appropriate cost estimation.

Some analytic systems completely rely on heuristics while others rely on a combination of heuristics and cost estimation to select the cheapest *QEP* for execution.

The cost of a query plan is expressed as a linear combination of intermediate result sizes (cardinalities) weighed by carefully defined factors for CPU cost, I/O cost, etc.

2.4.3 Cardinality Estimation

Cost estimation utilizes statistics of input data, the query, the cardinality of the data. The cardinality can be base relation or otherwise known as the table cardinality and the column cardinality. The defintion of cardinality given earlier in this section refers to the base relation cardinality. Column cardinality on the other hand refers to the number of distinct values in a database column or the number of distinct nodes in a graph.

The cardinality is important for several reason, one being that the cost model uses the information to estimate the cost of query plans. In fact, in many cost model architectures, the cardinality is usually the dominant factor in the cost function. In some ways the cost function keeps track of the changes introduced to the cardinality (increase or decrease) by performing certain operations at different stages in the query plan. This changes are estimated using what is known as the selectivity factor which is the number of tuples in the relation that satisfy a given predicate condition. Cardinality at different stages in the query tree are referred to as the intermediate cardinality or result size. In subsequent chapters, we will examine in details the cardinality and the selectivity of a query.

2.5 Recursion

In the the past years, there has been a resurgence in the interest and use of recursive graph queries. Recursion has it roots in deductive database and they are useful for expressing reachability properties or finding connections between connected data. One of the fundamental mechanism in modern data processing is recursion [52] and it has been applied and still useful in many domains and their use include finding the shortest paths, page rank, power and adjacency matrix, connected components, social network analysis in connected data which requires writing queries to traverse graphs. Recursion offers a good flexibility for describing relationships within graphs and performing computations on connected data.

Recursive queries expresses a category of complex queries that involves an iterative application of a function or operation until a certain predicate is satisfied — known as

the *fixpoint*. Several optimization techniques have been proposed for this class of queries including [17, 21–23] and more recently [4, 24–26].

EXAMPLE: Let us consider the popular transitive closure example; the ancestor-descendant relationship. Given a parents-children relation, the idea is to find the ancestor of each person and their descents as well. We can express this relationship using transitive closure algorithms.

Starting from an empty *ancestors-descendants* relation, the initial *parents-children* relation immediately becomes the first *ancestors and descendants* respectively. In the next iteration, the computation starts by looking for pairs of common nodes – yellow-colored rows *i.e.* any two nodes with a connection as shown in Figure 2.3. This process continues until no new results can be found and this point the computation terminates.

Parent	*Child*
Peter	Robert
Peter	Alake
Ola	Robert
Ola	Alake
Laura	Peter
Daniel	Laura

Figure 2.2 – Table describing parent-child relationship

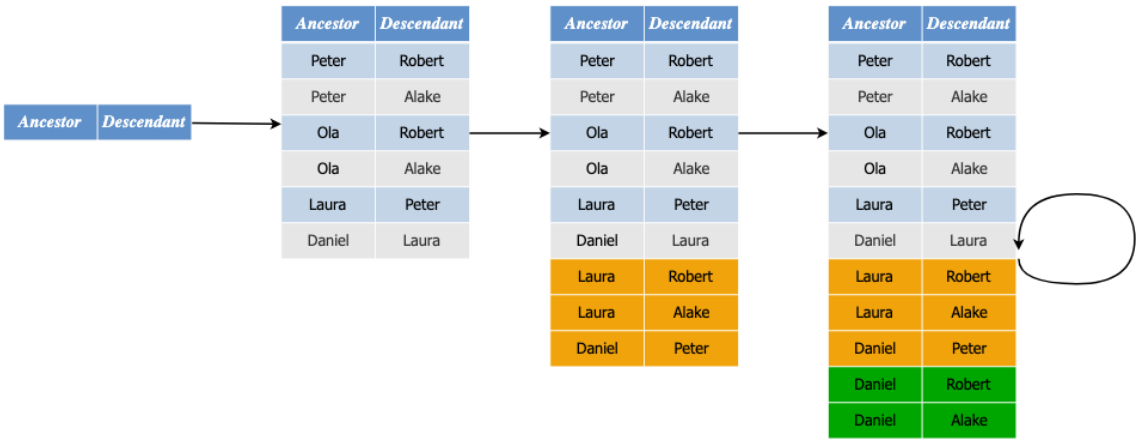


Figure 2.3 – ancestors-descendants relationship

Figure 2.3 shows the result at each step of the iteration. One could notice that the relation grows at each iteration, this is because new ancestor-descendant relations are being discovered from the ones of the previous steps.

2.5.1 Transitive Closure Algorithm and Evaluation

There are several algorithms for implementing the transitive closure which include naive, semi-naive, smart algorithm etc. In this section, we take a closer look at two of the most common and implemented transitive closure algorithms; naive and semi-naive algorithms. It should be noted that the type of transitive closure algorithm affects the evaluation time and cost.

2.5.1.1 Naive Algorithm

With *naive iteration*, results from the previous iteration are consumed in the next step of the iteration. As shown in Algorithm 1, naive evaluation involves performing repeated application of the function f on v until no new results are added.

Algorithm 1 Naive iteration

```
1: function naiveEval(F, S)
2:     while F(S) ≠ S do                    ▷ termination condition
3:         S ← F(S)
```

In line *3*, the operation is repatedly applied on *S* until the termination condition in line *2* is satisfied. This approach is inefficient as it does not eliminate duplicates which can be introduced during the iterations and this might lead to significant amount of data shipped across the network (in the case of distributed systems) or lodged in memory or disk.

2.5.1.2 Semi-naive Algorithm

Semi-naive evaluation of a fixpoint ensures that redundant re-computations are avoided, only new results generated in the last iteration steps are used in the current iteration.

Algorithm 2 Semi-naive evaluation

```
1: function semiNaiveEval(R)
2:     δ := R
3:     S := R
4:     while δ ≠ ∅ do                       ▷ termination condition
5:         δ := (δ ⋈ R) − S
6:         S := S ∪ δ
```

In Algorithm 2, an operator is applied only on the tuples produced during the previous iteration, δ is updated with the new results. In this manner, the algorithm avoids fully recomputing the join at each iterative step and the final result for a semi-naive algorithm is computed by taking the union of all the sets in its k^{th} steps in `Line:6`.

2.5.2 Fixpoint

Given an undirected graph G, containing a set of vertices V and edges E respectively, we define $G = (V, E)$ such that $E \subseteq V \times V$. Assuming $E(s,t)$ is a relation, there exists an edge y that is mutually reachable from x *iff* (x,y) is a tuple of E.

A fixpoint for an operator $f(s)$ iteratively evaluates the function f on s until a certain predicate is satisfied [4, 22, 53, 54]. Basically, the function f is repeatedly applied on v until no new result can be added to the solution at the n-iteration such that $s, f_1(s), f_2(s), ...f_{n+1}(s)$ represents each steps of the iteration. At the $n-th$ iteration, $f_n = f_{n+1}(s)$ at which point the computation terminates.

2.5.3 Algebraic Framework

Jachiet et al. [26] introduced an extended relational algebra called MuRA or φ-RA which implements novel optimization techniques by rewriting for fixpoint operators. The algebra presented corresponds to Codd's classical relational algebra extended with a fixpoint operator, and whose grammar is given as follows:

MuRA Grammar

$$
\begin{array}{lcll}
\varphi & ::= & & \text{term} \\
 & & R & \text{relation variable} \\
 & | & X & \text{recursive variable} \\
 & | & |c \to v| & \text{constant} \\
 & | & \varphi_1 \cup \varphi_2 & \text{union} \\
 & | & \varphi_1 \bowtie \varphi_2 & \text{join} \\
 & | & \varphi_1 \triangleright \varphi_2 & \text{antijoin} \\
 & | & \sigma_f(\varphi) & \text{filtering} \\
 & | & \rho_a^b(\varphi) & \text{renaming} \\
 & | & \widetilde{\pi}_a(\varphi) & \text{anti-projection} \\
 & | & \mu X. \underbrace{\varphi}_{\text{constant part}} \cup \underbrace{(R \bowtie \varphi)}_{\text{recursive part}} & \text{fixpoint}
\end{array}
$$

Since the only way to evaluate the effectiveness of a cost model is to integrate them into a query optimizer, MuRA framework is suitable for our work thanks to the advanced rewriting techniques which allows merging and nesting of fixpoints.

Description of Grammar

R relation variable for accessing the database relation

X variable that reference the recursive relation

$|c \to v|$ mapping of constant to variable

$\rho_a^b(\varphi)$ rename a to b

$\widetilde{\pi}_a(\varphi)$ set-based operator that removes column a from the relation

$\sigma_f(\varphi)$ eliminates tuples that does not satisfy the boolean condition f

$\varphi_1 \cup \varphi_2$ set union between φ_1 and φ_2

$\varphi_1 \bowtie \varphi_2$ combine compatible tuples from φ_1 and φ_2

$\varphi_1 \triangleright \varphi_2$ the set of tuples in φ_1 that are not present in φ_2.

$\mu X.\ \varphi \cup (R \bowtie \varphi)$ the fixpoint operator contains two parts; the constant part and the recursive part. They both consists of a set of operators that will be evaluated. However, the evaluation of the recursive part requires many steps until no further result can be obtained from the set of operators in it.

Additional definitions for all of the operators described here is given in Chapter 4 where we give the cardinality and cost estimation formulas for each one of them. We refer the reader to the work of [26] where the author elaborate on the grammar and rewriting techniques.

Chapter 3

An Overview of Cost Model in Query Optimizer: Towards a Recursive Query Cost Estimation

With several decades of research and development on query optimization, cost estimation still remains a central part of query optimization and they determine the quality of plan selection. In this chapter, we present a survey of cost models in existing systems by making a comparison between those systems and examine their support for recursive query cost estimation. We conclude the chapter by outlining the limitations and the directions we took for designing a cost model for the query optimizer.

3.1 Introduction

A query optimizer consist of three main components; the plan enumeration, cardinality and cost estimation which work together to carry out optimization process. One of the areas of query optimization that has been a subject of active research apart from optimization techniques are the cost and cardinality estimation. Query optimizers use the cost model to make accurate estimation of the cost of *QEP* [7, 32, 55, 56]. During query optimization, different alternative plans are generated and the optimizer is charged with finding the best plan or strategy to evaluate a given query. The alternative plans generated by the optimizer differs in terms of their runtime cost from a few seconds to several minutes and even hours. And as a result this, the cost estimation has significant influence on the quality of plans selected by the query optimizer and the query performance. The cost model computes the cost for each *QEP* in the plan space and the optimizer selects the plan with the minimum cost for execution.

Cost estimations are particularly useful in selecting the best join order that improve query performance [7, 57], determining the appropriate algorithm for executing operators [7, 55], query performance prediction [58–60] and query scheduling and progress monitoring [32].

In section Section 3.2, we present the challenges that are often faced when estimating the cost of a query plan. We present an overview of cardinality estimation and methods for statistics collection for cost model in section Section 3.3. Cost model and their classifications is discussed in Section 3.4 where we classify cost models based on their architecture, plan style and graph-centric cost model. A set of criteria were defined and on that basis, we compared cost estimation techniques of state-of-the-art systems ranging from traditional databases to distributed big data systems to graph database in Section 3.6. We summarize our findings in Section 3.7.

3.2 Challenges of Cost Estimation

There are several factors that could affect the accuracy of cost estimation in database systems. In this section, we present some of those challenges.

- **Advancement in software-hardware:** database systems and their underlying software and hardware platforms become increasingly sophisticated [58], improvement on the state-of-the-art in cost model are often affected by the advancement in hardware. Early cost models [7] were made for centralized architectures. [61] gave a generalized cost model for the distributed setting but it neglects interaction between hardware and the different components of the database.

- **Estimation error:** one of the biggest challenges of query plan cost estimation is a bad cardinality estimation [35, 55]. Estimation error comes from two part of the cost model. First, errors can be introduced as a result of poor cardinality estimation. Second, the estimation error can originate from the cost function. Because the cost function uses the cardinality as part of its input parameter, if there is any error generated as a result of cardinality estimation, these errors are introduced and propagated to the cost function. In some cases, the errors are mitigated in the cost function resulting in the selection of near-optimal plan for execution. In other cases, the result of propagating these errors can be catastrophic resulting in the optimizer's selected plan (according to the cost function) being a very bad plan. Some of the reasons for inaccurate cardinality estimation includes inaccurate base statistics, the presence of multiple join relationships, inaccurate selectivity estimation, bad assumption about the distribution of data etc. Many database systems including commercial ones suffer from this problem of estimation error propagation.

- **Dynamic or Changing Job Parameters:** in a dynamic environment where job paramters and data changes during query execution, relying on static paramters can

introduce significant errors into the cost estimation.

- **Complex Job profile:** this characteristic is specific to the distributed setting, the cost parameters are either too complex [31, 59] or too simple [62] and do not cover other necessary factors or parameters that can influence query performance like the underlying hardware info or cluster description at runtime. Another example of complex job profiles can be found in learning-based approaches [58, 63] where different job paramters like execution plan, actual query cost and training (historical) datasets are needed to efficiently predict query runtime.

- **Complex queries/Limited operator support:** cost estimation for multi-join queries involving several database tables is often difficult and error-prone, since the presence of local predicate on any of the participating relations affect the intermediate result size. If this is not properly taken into consideration during cardinality estimation, the local predicate on participating relations can introduce error significant enough to cause the query optimizer to select a bad plan. In addition, recursive queries has made it way to mainstream databases allowing for the formulation of complex queries on nested data structures. RDBMS systems lack native support for the recursive operator but rather support only the limited form called linear recursive queries as a recursive CTE. While the support for simple recursion is now common in many known database systems like Oracle, Postgres, MS SQL etc, more complex forms of recursion (or non-linear recursive queries) like fixpoint operation are verebose and often formulated in a WHILE loop. We note that query plan cost estimation on these systems mostly covers SPJ (Selection, Projection, Join) family of SQL queries which is sufficient to a certain extent for queries that does not involve recursion.

3.3 Cardinality Estimation

Relation cardinality is not the only parameter or input statistics used by the cost function to estimate the cost of a query operator and query plan at large but perhaps the most discussed and studied because of their relevance in selecting quality query plans. We take a look here at the research on cardinality estimation in *RDBMS* and graph database.

In Figure 3.1, we classify cardinality estimation into three broad categories; sampling-based, synopsis and learning-based methods which are cardinality estimation methods with originated from RDBMS some of which are also applied to graph data. Synopsis method is further classified into three; histogram, sketches and other methods that do not fit the first two. Graph-centric cardinality estimation methods are only applicable to graph data. They mostly focus on data sampling algorithms and reducing the complexity of RDF data by summarizing the data into more concise form for an easier estimation.

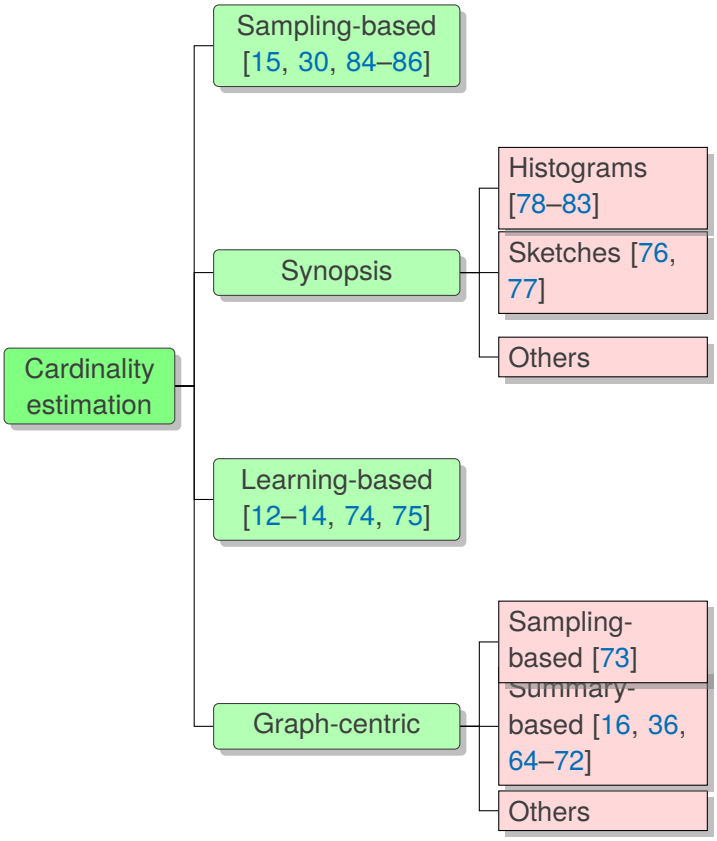


Figure 3.1 – Cardinality estimation

3.3.1 Statistics and Methods of Collection

Cost estimation in query optimizer rely on a number of parameters in order to make better estimation. Perhaps, two of such factors include cardinality and selectivity estimation. Accurate cardinality estimation has been demonstrated to have a significant impact on the quality of plans selected by the optimizer [32, 38, 41, 55]. Cardinality estimation is an important factor in query optimization and typically relies on heuristics and basic statistical approximations [39]. Cost input parameters can be categorized into two; the database profile and the amount of available system resources. The database profile contains (1) the catalog statistics which holds basic information of data properties like relation size and the number of distinct attribute values, index, block size, size (in mb) of the data etc. (2) query-specific information like the cardinality and selectivity [87]. The system resources paramaters include the amount of memory allocated (or available) for the execution, the number of nodes and cores, the replication factor etc. Total *QEP* cost returned by the query optimizer are usually a combination of the different cost components.

Collection of statistics for the cost function is done prior to query planning and execution as statistics collection is an expensive process that will eventually affect significantly the query execution time. Statistics collected on data are stored in a database catalogue or filesystem and accessed during cost estimation. In distributed settings, resources are often specified just before the execution and this indicate the amount of system resources

usable by the running application. These informations are also used by the cost function since applications behave differently depending on the resources. At execution time, base relation cardinality is known and propagated through the query tree [10]. For the cardinality of the other operators apart from relational variables that access the cardinality catalogue statistics directly, the input cardinality is given by the output cardinality of the child node in the query tree [88]. When a query involves joining relations, this will affect the intermediate result size (or cardinality) and also have significant influence on the chosen *QEP*. When a *filter* operation is applied to a relation, only the tuples that satisfy the given conditions will be returned. This property is referred to as the *selectivity*.

3.4 Cost Model

In the section, we define a cost model and give an architectural view of a traditional cost model in both traditional database system and distributed big data platforms.

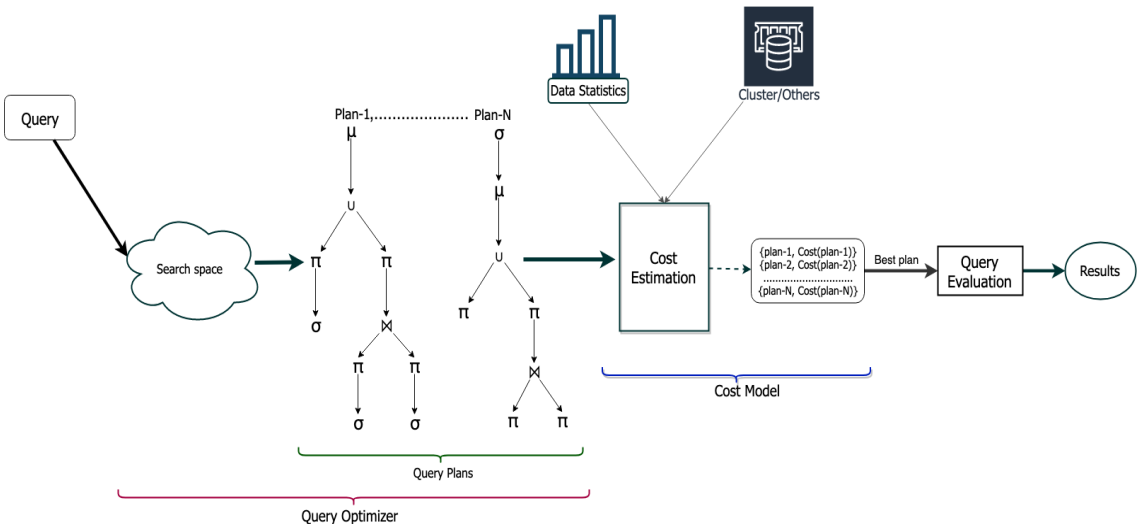


Figure 3.2 – Cost model architecture

Figure 3.2 show a cost model architecture. The cost model accepts a set of query plans, input statistics, cluster parameters (in the case of distributed systems) and estimate the cost for each plan and output *QEP* with the cost attached. The optimizer then selects a plan with the minimum cost. As we will see in the next section, cost models have a tight relationship with the architecture of the underlying system and in some cases the query plan types.

In most traditional database systems, the cost is usually a summation of the CPU and I/O cost [7, 32] since they are based on the main-memory architecture and do not involve transferring data over the network. Cost models in the distributed setting accounts for for message passing cost over the network anytime such an operation is performed. I/O time is the time spent accessing the disk while the CPU cost is the weighted CPU time.

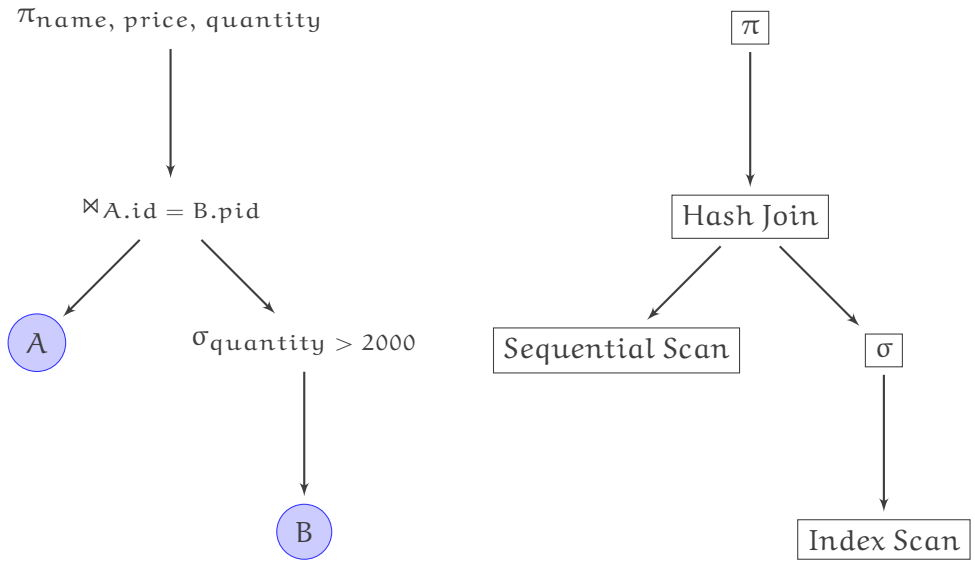
3.4.1 Cost Model By Plan Enumeration Style

In a query optimizer, there are several levels of optimization and broadly they are divided into logical and physical optimization. The difference between these two is that logical optimization often deals with the process of generating optimal sequence of equivalent relational expressions or subexpressions for a query while physical optimization is concerned with finding the best algorithm to implement the logical sequence of operators and the order in which the physical operations are performed. For example the implementation details of a join operator like using a hash join or nested loop join is an information that is available only at the physical level. This distinction between optimization level gives rise to the (i) logical and (ii) physical cost models.

3.4.1.1 Logical Plan Cost

A logical cost model is simply the series of cost functions applied to logical query subexpressions. They are applied on the relational algebraic operator and the goal is to determine which is the best combination of operators of the query tree and in which order to execute them. In many cases, the logical cost considers only the data distributions and the operational semantics of the relational algebraic operations to estimate intermediate result sizes of a given logical QEP.

Figure 3.3a shows an example of a logical query plan. Usually the logical plan cost function in many database systems rely solely on the cardinality estimation (and intermediate result size) to determine the best combination and predict the efficiency of the logical query plan.



a Logical query plan G

b Physical query plan

Figure 3.3 – Query Plans

3.4.1.2 Physical Plan Cost

Physical query plans contain the logical query plan with the addition of annotations on the access path with which the relation is accessed, algorithms used to implement each operator and the order in which the operations are performed. Cost functions for the physical plans usually take this information into consideration. A cost model for the physical *QEP* usually encompass the information about the CPU speed, I/O latency, I/O bandwidth, and network bandwidth [88], especially for systems that use more complex cost models, accounting for almost every possible scenario and paramter values. For example, there are different types of join algortihm e.g. Nested Loop Join, Merge Join and Hash Join, the cost of using any of these join strategies depend on a variety of conditions like the intermediate result size, join ordering in the initial query statement, etc. An example of physical query plan is shown in Figure 3.3b.

3.4.2 Cost Model By Architecture Type

Cost models can also be categorized by architecture type. Most relational database systems like Postgres, MySQL, Oracle, and many others are main-memory architecture-style databases and they are referred to as centralized database system; computation happens mainly on disk. On the other hand, we have the distributed processing systems like Spark, Flink, Presto etc. that allows data and computations to be distributed across several worker nodes making them scalable and allowing for faster computation. We discuss the cost model for these two architecture in the next section.

3.4.2.1 Cost Model for Centralized Database

In a traditional database system, data storage and computation are centralized such that whenever a user-defined query is executed, the query engine access the data and perform its processing by accessing a single data location and returns the result avoiding any need for movement of data across the network. This approach offers a cheap alternative and avoids any need for data to be made redundant.

Query plans' cost estimation for this type of architectures has been based on the work of Selinger et al. [7] which defines the cost functions that recursively (in a bottom-up fashion) estimate the CPU and I/O cost for each operator in a query tree starting from the relational variables. A final combination of these two cost components for all operators in a query tree gives the cost of the entire query plan. Many *RDBMS* like Postgres, MySQL, Oracle etc. implement and use this type of cost model. The cost model is shown in Equation 3.1.

$$\text{total cost} = \text{I/O} + w * \text{CPU} \tag{3.1}$$

I/O is the disk I/O, CPU represents the CPU utilization cost and w is an adjustable weight factor that represent the balance of importance between and CPU and I/O.

3.4.2.2 Cost Model for Distributed Settings

Distributed data analytic systems like Apache Spark, Flink and Hadoop are designed in such a way that data reside or is distributed across several machine connected through a network and computations are either done locally first and the results from the individual worker nodes are merged in a central node referred to as the master. In some cases computations are carried out by requiring data with similar characteristics to reside in the same worker node before an actual computation is performed leading to data movements across the network.

Resources and a set of paramters are allocated to any running computation. These set of paramters are described in Section 3.3.1 and the resources include the number of worker nodes, the number of cores etc. As a result of this architectural design, cost functions for operators on these systems must reflect these characteristics. [61] present a cost model for distributed evaluation of queries on R* system a successor of the popular System R [7]. Their cost function combines four cost components; the CPU cost, I/O cost, the total number of messages (sent and received) and the total number of byte transferred in every message.

$$\text{Total cost} = \underbrace{\text{CPU} * \#insts + \text{I/O} * \#\text{I/Os}}_{\text{local computation cost}} + \underbrace{\text{MSG} * \#msgs + \text{TR} * \#bytes}_{\text{communication cost}} \tag{3.2}$$

CPU is the time spent executing the CPU instructions and I/O is the time spent for disk I/O, combination of which forms the local computation cost. MSG is the time spent initiating and receiving a message and TR is the time spent sending bytes of data from one node to another. The combination of the last two cost component is referred to as the communication cost.

Equation 3.2 is often rewritten as three components as opposed to four where the communication cost components are often combined to form the Network cost. Herodotou [59] present a cost model for the execution of MapReduce job on Hadoop based on three cost components; the CPU, IO, and Network costs. Similarly, [31] modeled the cost in terms of read, write, shuffle and broadcast.

3.4.2.3 Cost Model for Graphs

We explained in the previous chapter that graphs are a natural data structure when it comes to representing relationships between data. They are very powerful in this regard and have seen a resurgence in their use in the recent years. The standard language for extracting and querying information from RDF graphs is SPARQL query language. SPARQL can be translated based on the relational algebra of RDBMS, represented in a similar select-project-join queries in the relational model [89] but the underlying RDF graphs are represented in a SPO (Subject-Project-Object) pattern where subject S has property P with value O, where S and P are resource URIs and O is either a URI or a

literal value. Cost estimation techniques for SPARQL query plans are usually based on the underlying storage system and the complexity of the query.

3.4.3 Cost Estimation Strategies

As mentioned in the previous chapter (see Section 2.4.1), there are two widely adopted strategies for query optimization (i) top-down and (2) bottom-up. Even though these are different optimization strategies, the cost estimation is done in the same bottom-up fashion for these two strategies since the total cost of each plan is based on the sub-plan costs.

The estimated cost returned by the cost model in many database systems is usually a tuple of cost components e.g. estimated cost, cardinality, byte transferred. which are computed for every operator. Even though cost component types returned per operator are the same, some operators return no values for some components. For example, [59] defines a cost for queries on distributed; Hadoop which returns the network, CPU and IO costs. The filter operation for example does not have a Network cost as computation is transformational and local to the each worker, thereby having no Network cost.

Algorithm 3 : Cost estimation algorithm

Input: query execution plan qep

```
1:  function costEstim(qep)
2:      while (child ∈ qep.node) do
3:          if (qep.child = baseRel) then          ▷ baseRel refers to base relation
4:              stats ← retrieveBaseStats(qep.child)
5:              cost ← apply cost
6:              costEstim(child) ← (cost, stats)
7:          else
8:              prev ← costEstim(child)
9:              apply cost
10:     return cost(qep)                                  ▷ Total QEP cost
```

During query plan cost estimation, the *QEP* is traversed in a bottom-up fashion and cost estimation is done recursively. The cost of each operator in the query tree is estimated using an engine-specific cost formulas defined for each operator. Algorithm 3 shows the cost estimation algorithm, the function accepts a query plan and data statistics and compute the cost for each child node. For relational operators (index or sequential scan) which access the relation directly, the cost formula uses the base relation statistics and this information is propagated to the other child operator as the intermediate result which is later used for its own estimation in `Line:9`.

3.5 State-of-the-art

In this section, we review some previous works on cost models for query optimization. We broadly classify work in this sphere into three categories; (i) works on **improving existing cost and plan quality** (ii) **performance prediction** (iii) insights paper; which give new perspective to cost estimation in query optimizer. This classification spans across the types of cost model described in Section 3.4, ranging from cost model for logical and physical query plans to centralized, distributed and graphs. We discussed some representative works for each category.

3.5.1 New or Improvement on Existing Cost Model

In this section, we examine the research works on new methods for cost estimation for both centralized and distributed environments, for both RDBMS and graph models, for either logical or physical plans. We begin this section with early works on cost estimation in query optimizer of the R Systems [7, 61] spanning both centralized and distributed architectures. The discussion here also covers cost models in commercial database systems, graph models and works that extends existing framework with the objective to improve the quality of query plan selection.

System R [7] introduces a query optimizer for SQL queries. The cost covers the the *SPJ* (selection, projection, and join) family of SQL queries. In order to compute the cost and subsequently optimize the query plan, the optimizer uses statistics maintained on relations and access path, predicate in the query, the access paths available on the relations and an query order e.g, ORDER BY or GROUP BY. By using statistics of input data and assigning a selectivity factor for each boolean operation in the predicate list and a set of simple assumptions, they estimate the cost for query plan in a centralized manner using a bottom-up approach. The cost returned by the computation is in terms of the IO and CPU cost. The cost model assumes that CPU cost is mostly negligible and so a weighted factor (W) between the IO and CPU was assigned. The cost formula is given in Equation 3.3

$$\text{Cost} = \text{cost}_{\text{IO}} + W \cdot \text{cost}_{\text{cpu}} \tag{3.3}$$

While the cost model described here is not entirely suitable for distributed setting since it does not account for the cost of message passing and data shipping over the network, it has set precedent for most modern cost models and has even been adopted by modern query engines. The cost model described in this work makes an assumption, the uniform distribution of attribute values.

PostgreSQL query optimizer uses a cost-based approach to select efficient query plans among the possible list of alternative physical query plans for a given query and also uses relation and column statistics as well as histograms for selectivity. The statistical information that is used by the cost model is stored in the pg_statistic of the database catalogue or pg_stats that provide public access to the information stored in pg_statistic. To estimate the cost, PostgreSQL computes the CPU cost of an operator in the query tree

by using the cardinality estimate and compute the IO cost as a function of the estimates of the the number of pages accessed. Cost estimation in PostgreSQL follows the formula given in [7] and the CPU and IO costs are summed to get the final cost [32, 55].

The cost of an operator, C_o in PostgreSQL [32] is given as;

$$C_o = n_s.c_s + n_r.c_r + n_t.c_t + n_i.c_i + n_o.c_o \tag{3.4}$$

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

And c_s represents the cost of sequential page access, the cost of random pages access is given as c_r. c_t represents the cost of processing a tuple, c_i the cost of processing a tuple via index access and c_o is the cost of performing hash or aggregation operation.

Catalyst [62] is an extensible cost-based optimizer for Spark SQL which allows for easy addition of optimization techniques ontop of Spark SQL. Spark SQL performs cost-based optimization both on the logical and physical level. The logical cost model is done by generating multiple plans using rules, and then computing their costs. The logical cost estimation is based on the number of output rows in the result relation. The physical cost model is only used to select join algorithms. Figure 3.4 shows the Catalyst optimizer

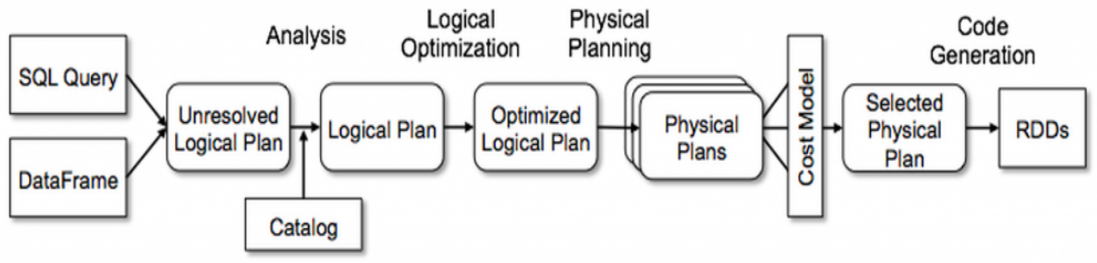


Figure 3.4 – Spark Catalyst optimizer architecture

architecture. The cost-based optimization is carried out in a series of steps in Spark SQL; collect, infer and propagate relation statistics (number of rows, table size in bytes) and column statistics (e.g. distinct and null count, average and max length, histogram etc.) on source or intermediate data. The cost per operator is estimated in terms of the number of output rows, output size, etc., based on the cost estimation, the best query plan is then selected for execution. The cost function in the Spark Catalyst optimizer also follows the work of [7].

Golfarelli et al. [31] develop a cost model that measures query execution time for SQL queries on Apache Spark SQL platform. The cost model covers the Generalized Projection, Selection and Join SQL operators (referred to by the authors as GPSJ) family of SQL operators. The cost estimated is based on disk access, network time and cpu (serialization and compression) time. It models the execution plan in terms of the costs of the physical plan returned by Spark Cost-Base Optimizer (CBO) – Catalyst. They define the execution time as the time needed to execute nodes of the trees coding the physical plan produced by Catalyst. Operations in a GPSJ query is categorized into "read", "write, "shuffle" and "broadcast". Using application and cluster configuration parameters, the statistics of the input data, they present a cost model tailored only for query execution on the Apache Spark platform.

In the COBRA [90], the author presents a framework for rewriting data processing programs using program transformations. The author defined a cost estimation for individual nodes in Region DAG of algebraic expressions based on the Volcano/Cascade framework.

Sun and Li [63] proposed an end-to-end learning-based estimator for estimating the cost and cardinality of query operators and plans using a tree-structured (deep neural network) LSTM model. The learned cost estimator is composed by three components; (i) training data generator for generating varying training data based on data and workload, (ii) feature extractor for extracting meaningful features from the query plan, and (iii) tree-structured model which learns the tree representation of query plan effectively by matching sub-plans to sub-model.

Obermeier and Nixon [91] presents a cost model for distributed SPARQL processing. Since SPARQL algebra can be translated to the relational algebra, the authors mapped the techniques for cost estimation from the RDBMS domain to estimate the cost in a SPARQL graph query model. The cost focuses on three components, the number of CPU instructions, I/O operations and network delay for data retrieval. CostFed [92] presents an index-based SPARQL federation engine that uses cost selectivity information stored in an index to make efficient source selection and cost-based query planning. The cost model proposed there considers the skew in the frequency distribution for triple patterns.

Errors introduced by inaccurate cardinality estimation significantly affects quality of selected query plans. This is because the cost model rely on the cardinality estimation for computing the cost of the different alternative query plans for any given query among which the cheapest plan is selected. In practise, this errors in cardinality are propagated in the cost function meaning that the cheapest plan selected by the optimizer could potentially be a significantly suboptimal plan. Wolf et al. [93] present three metrics that quantify the robustness of query execution plans at optimization time and consider the potential implications of errors introduced from the cardinality estimation during plan selection. The authors presents their robust plan selection technique in three phases; (i) enumeration of robust plan sets, (ii) computation of the robustness value for each candidate query plan, (iii) selection of the most robust query plan among these candidate plans.

Lanzelotte et al. [23] introduced cost-controlled strategies for optimizing recursive object-oriented graph queries. The authors propose an optimization method that relies on a cost model for selective pushing of operators through recursion. The authors argue that for efficient query plan optimization (especially when objects and recursion are involved) and rewriting, a cost model is needed to measure the impact of each of the optimizer's actions.

Fixpoint ($Fix(T, P)$) cost deserves a special mention here as the authors give the cost for a semi-naive TC algorithm as follows;

$$Fix(T, P) = \sum_{i=0}^{n} cost(Exp(T_i))^c \tag{3.5}$$

where n is the number of iteration, $Exp(T_i)$ denotes the fixpoint equation contained in P having T_i input.

The cost of PT node rooted at N such that $N(child_0, child_1, ...child_{k-1})$ is given as;

$$cost(PT) = cost(N) + \sum_{i=0}^{k-1} cost(child_i) \tag{3.6}$$

Ioannidis [21] presents algorithms for computing transitive closure (TC) of the relational operators. In order to compare performance of several TC algorithms they devised an I/O cost analysis use of simpler set of statistics to estimate the cost for each TC algorithm. I/O cost analysis for semi-naive transitive closure algorithm is given as follows;

1. Sort original relation on appropriate field(s). It is done only once.
2. At each step read sorted original relation
3. Sort the second relation for the join
4. Write the outcome of the join
5. At the end read all the intermediate results and put them into one relation.
6. Create and Destroy the N intermediate results and also create the final result.

3.5.2 Performance Prediction

Work on performance prediction covers works on cost model tunning and performance prediction. And they also include works that use machine learning, deep learning and AI to predict the behaviour of query workload.

To tackle uncertainty in the runtime behaviour of a query – execution time and resources consumption, Ganapathi et al. [60] present performance metric model for predicting query performance. Using statistical relationships, their goal is to find correlations among the query properties and query performance metrics on a training set of queries to predict performance of future workloads. These features are then used by the machine learning

technique to learn appropriate cost functions for each operator based on the system configuration.

In [58], Akdere et al. present a learning-based approach to predict the query performance. The predictive model learn the query execution time at both plan-level and operator-level where feature and performance values are extracted.

While many research work are focusing on finding alternatives to traditional cost model by adopting (machine, deep) learning methods, Wu et al. [32] argued that instead of treating query optimizer cost model as a blackbox, the information about the query and optimizater can be used to accurately predict query execution time. The authors proposed a cost model tunning approach that uses (i) hardware profiling and cost units calibration and (ii) cardinality sampling method for obtaining operators true cardinality per query, to predict query execution times.

Herodotou [59] described a cost model for MapReduce job execution on Hadoop. The model described the cost for a MapReduce job at a finer granularity while capturing the different phases for both map and reduce tasks. The sum of the cost from each of the task phases represents the overall (total) cost of a MapReduce job. The cost component tracks the CPU, I/O and Network costs using the general formula proposed in [61]. The paramaters used for costing query plan are divided into three; (i) hadoop parameters which define the set of hadoop-specific parameters that effect job execution, (ii) profile statistics which is basically the input data statistics and the properties of the user-defined functions and (iii) profile cost factors which are a set of parameters that define the CPU, I/O and Network cost. The goal of the cost function is to predict performance and find optimal settings for a MapReduce jobs on Hadoop. It should be noted however, that some of the paramaters used by the author are rather too rigid and complex.

COMET [94] present a statistical learning technique for query performance prediction on semi-structured data: XML. The authors adopted a four-step approach to costing XML operators; (i) identification – of important determinants features (e.g. as algorithm, query, and data) of the cost (ii) feature value estimation using statistics and analytic formulas (iii) learn relationship – between cost and feature values using machine learning algorithms (iv) application – of the learned cost for optimization.

Hasan et al. [95] present a machine learning approach for predicting SPARQL query performance. The approach presented there learn from historical information about query execution (query execution times only) and apply machine learning to predict the query execution times for future workloads with similar properties.

3.5.3 Experimental or Insight Studies

In this section, we present some representative works on cost models that shed more light into the main problems encountered in existing systems and proposed solutions to solve them.

3.5.3.1 How Good Are Query Optimizers, Really? [55]

In this work [55], the authors started by decomposing traditional query optimizer architecture into several components. These components were studied in isolation (and also together) and their effect on query performance was presented.

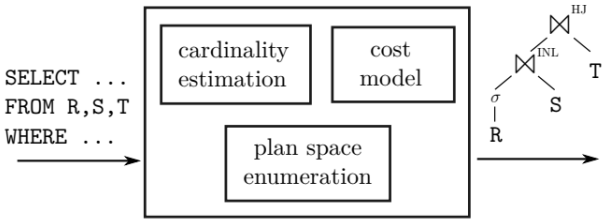


Figure 3.5 – Traditional query optimizer architecture

A Job Order Benchmark (JOB) was introduced which uses a realistic dataset that diverges from the simple "uniformity", "independence" and "principle of inclusion" assumptions in order to answer the following questions;

- How good are cardinality estimators and when do bad cardinality estimates lead to slower queries?
- Of what importance is an accurate cost model to the overall query optimization process?
- How query plan space must be?

The authors performed extensive sets of experiments on the different components of the query optimizer using the following setup;

- **Dataset**: realistic dataset (3.6GB consisting of two large tables each of 36M and 15M rows rsp.) which are full of correlations and non-uniform distributions of values.
- **JOB queries**: a 33–structure query set totalling 113 queries consisting of an average of 8 joins each, analytically constructed.
- **Database System**: a total of five(5) database systems were used to validate some of the propositions. Being open source, Postgres was used at all stages of the experiment.
- **Cardinality extraction and injection**: In a first step of the experimental evaluation, statistics gathering command was run on each database system and cardinality estimates for all intermediate results were obtained. Also, the true cardinalities for each intermediate result was obtained and later used to obtain optimal query plans with respect to each system. Lastly, PostgreSQL was modified to manually inject true cardinalities for arbitrary join expressions allowing an effective measure of query performance.

Cardinality Estimates

Cardinality estimation plays an important part in query optimization. The authors noted that cardinality estimate remains the most important factor for obtaining a good query plan arguing that its importance outweighs that of cost model and plan space enumeration. Cardinality misestimation (misestimation here is under- and over-estimation) is an inherent problem in many database systems and these misestimations are more pronounced in query optimizers that assumes independence and correlation between data. Their findings showed that query optimizers that uses *data sampling* produces estimates that are closer to the true values. Cardinality misestimation increases as the number of join grows, however, this misestimation does not necessarily mean bad query performance.

Misestimation and Slow Queries

Cardinality misestimation does not necessarily mean a bad query performance, further investigations on when misestimation can lead to bad query performance was achieved. Since query optimization is closely intertwined with physical database design, the type and number of indexes, plan search space and subsequently cardinality estimation.

It was discovered that one query optimizer selects join algorithm which are more expensive. For instance, PostgreSQL introduces nested-loop join instead of an index lookup with a cardinality return a way smaller than the true one. A purely cost-based optimizer which does not takes into account the complexities of the chosen algorithm and the uncertainty of cardinality estimates can lead to bad plans. The findings concluded that in main memory - where index and data are fully cached, picking an index-nested-loop join over a hash join will be better in terms of performance.

Cost Model

From different alternative plans in the search space, the cost model guides the plan selection. The role of a cost model is to predict which of the alternative query plans will be fastest, given the cardinality estimates. The authors noted that cost models even in modern database systems are mostly traditional disk-based.

By introducing a simple cost model and tunning the cost model of the database systems discussed here to use true cardinalities. The following was reported;

- Cost & Runtime: it was reported that PostgreSQL cost model produces a high standard error with so many outliers because it misestimates cardinality. When true cardinalities were injected, it was noticed that PostgreSQL cost model produces a more reliable estimation of the runtime with little outliers. It was also mentioned that *most database systems always assign higher cost for expensive queries.*

- Tunning the cost model for main memory: default parameters of cost models for most database systems are sub-optimal e.g. PostgreSQL suggest that page read is $400x$ $times$ more expensive than processing a tuple. By making the CPU cost parameters of the PostgreSQL cost model $50x$ $times$ more expensive, it was observed

that there was an improvement in the correlation between the cost and the runtime. However, by using the true cardinalities, there was a significant improvement in the cost and runtime.

- Complex cost models: to verify if complex cost models are needed to properly estimate costs that have high correlations with the query time, the authors introduced a very simple cost model and compare the query time with the original cost from the PostgreSQL cost model. This very simple cost model outperforms PostgreSQL cost model using the true cardinalities with a 41% improvement.

The experiments in this section further confirms the author's claim that cardinality estimation is more important than the cost model for main memory database systems.

3.5.3.2 Notes on Cost Modeling [96]

Wu [96] present a study on the robustness of operator-level cost modeling. The author noted three challenges in current cost modeling using query execution feedback which include; (i) lack of appropriate training, (ii) unavailability of sufficient training data, and (iii) difficulty in combining learning-based approach with default optmizer's cost. To address these challenges, a framework that operates on a limited execution feedback scenario was proposed consisting of three steps;

- identification of backbone leaf operators; in this case able scans, index scans, and index seeks
- using existing techniques, build an external cost model for each leaf operator in a query plan
- combination of the query optimizer's internal operators cost estimates with the external cost model for leaf operators

Results from detailed experimental studies shows the effectiveness of the technique. This study presents a new strategy for mixing operator cost estimates using sparse feedback.

3.6 Cost Classification Criteria

In Table 3.1, we summarize the various features of existing cost model in state-of-art systems. We discuss each criteria below. The first column represents the system or paper under study, a total of 16 systems/approaches were considered; 12 in centralized and distributed setting and 4 learning-based approaches.

- **Settings:** given the components of cost models (whether they accounted for the network or not), we have categorized them as either *centralized* or *distributed*. As metioned earlier in this chapter, a graph-centric cost model is also dependent on the underlying storage system which can be either be centralized or distributed.

- **Time function:** the cost returned by the cost model is either a measure of the time in seconds – absolute time for the query evaluation or a score which represents how expensive the query evaluation will be – relative time, mostly the lower the better.

- **Cost component:** cost formulas are made up of components that are summed up to get the cost of each operator in a query tree. We explained that this components are either CPU, I/O, Network (or communication) cost. In a centralized setting, the cost components only consist of the first two, the later is specific to distributed environments.

- **(Input) Statistics:** the statistics available for computing the different components in the cost model. These statistics ranges from `table` (row count, column distinct count and average length of tuple), index (number of distinct index, number of pages in index), histogram on the distribution of tuples in the relation and in the distributed setting we have cluster parameters and network-related constants.

- **Cost type:** based on the classification in Section 3.4, the type of cost models considered are either physical or logical cost models. Catalyst [62] has both physical and logical costs just like many distributed systems. The logical cost is based on the number of output rows by each operator. And the physical cost model in Catalyst is used for join order re-optimization.

- **Query type:** represents the set of operators captured under the cost function for each system. There are four common types of operators of interest here: **S**election, **P**rojection, **J**oin and **R**ecursion (transitive closure or fixpoint) operators. Specifically, R column indicates whether the system under study has support for specialized recursive cost estimation function. Of all the systems considered, only two [21, 23] have the support for cost estimation for recursive operators. [21] only focus on transitive closure, [23] defines the cost for query processing tree which includes a cost formula for fixpoint operator. All the other systems only consider cost functions for the SPJ family of queries.

3.7 Our Findings

In Section 3.4, we examine numerous cost estimation techniques in traditional relational database system, distributed and the graph domain. These methods or techniques either focus on improving existing cost models, present a new way of estimating the cost for query plans or present a study to gain insights into already proven methods. An important aspect to note is that, although the work of [7] is focused on the centralized cost estimation of query plans, it is the foundation for cost estimation in many database systems, including commercial ones like PostgreSQL, Oracle etc. Mackert [61] extended

this work for distributed settings, accounting for the response time of query with the introduction of additional cost paramters; the communication cost.

3.7.1 Support for Recursive Cost Estimation

We found that cost models for recursive operator support and user-defined functions in mainstream database is still lacking. Learning-based approach, while they are not mature enough for query performance prediction also suffer similar drawbacks as in existing systems. Cost-based optimization in system that rely on the techniques discussed so far are presented with the difficulty of optimizing the recursive operator. In Table 3.1, only two [21, 23] out of the 16 systems considered cost estimation for recursive operators.

Lanzelotte et al. [23] present an algebraic (logical) cost model for query plans with the fixpoint operator. The cost of the fixpoint operator is, however, over-simplified as it does not account for the true behaviour of the operator. This raises many questions like how does the cost model tracks the changing result size at each iterative steps, when does a recursive operation terminates or ends. The fixpoint operator is evaluated in such a way that at each step, new results are added to the facts already derived until there is no new results that satisfy a particular condition.

Similarly, the I/O cost analysis of Ioannidis [21] for semi-naive and smart transitive closure algorithms focuses on binary relations. For the evaluation of the cost analysis, the author rely on a simulation of the cost paramters. In addition to not reflecting the real-world scenario, some implementations of the recursive algorithm can permit nested recursion as in [26] where the fixpoint operator can be nested.

3.7.2 Possible Future Directions

The cheapest query plan selected by an optimizer's cost model can differ significantly from observed more efficient ones. Current methods of evaluating the effectiveness of a cost model by comparing the query runtime is not enough to give a clearer view of what happens under the hood of a cost function. Instead of comparing the cost models based on the query time of the selected plans, a cost validation framework that takes into account the estimation errors (cardinality and cost) and query plan ranks, will help shed more lights on the behaviour of the cost model.

Support for recursive cost estimation is crucial in todays database systems. As a first step, more and more systems need to provide native support for the recursive operator. Many distributed analytic frameworks like Spark, Flink and MapReduce already offer a relatively fast way of evaluating queries but they are still lacking support for recursion.

Learning-based methods for query optimization is an active area of research. Model training time and data to be used for training are two of the biggest obstacles in using learning-based techniques. A standard cost feature repository will help in the long run in terms of feature collections and historic information providing more data for evaluating learning-based techniques.

Table 3.1 – Summary of state-of-the-art cost model

System	Settings	Cost components	Statistics	Cost type	Query support				Time function
					S	P	J	R	
System R [7]	centralized	CPU, I/O	table, index, column, sFactor	physical	✓	✓	✓	X	rel. time
PostgreSQL	centralized	CPU, I/O	table, index, column, sFactor	physical	✓	✓	✓	X	rel. time
R* [61]	distributed	CPU, I/O, #Msgs	table, index, column, sFactor	physical	✓	✓	✓	X	rel. time
Herodotou [59]	distributed	CPU, I/O, Net.	hadoop parameters, profile statistics & cost factor	-	✓	✓	✓	X	abs. time
Catalyst [62]	distributed	CPU, I/O, Net.	table, index, column, sFactor	logical, physical	✓	✓	✓	X	rel. time
Golfarelli et. al [31]	distributed	CPU, I/O, Net	table, column, cluster	physical	✓	✓	✓	X	abs. time
COBRA [90]	distributed	CPU, I/O, Net	table, column, cluster	physical	✓	✓	✓	X	abs. time
CostFed [92]	distributed	CPU, I/O, Net	table, column, cluster	physical	✓	✓	✓	X	abs. time
Obermeier et al. [91]	distributed	CPU, I/O	table, column, cluster	logical	✓	✓	✓	X	rel. time
Ioannidis et al. [21]	centralized	I/O	table, column	physical	X	X	✓	✓	abs. time
Lanzelotte et al. [23]	centralized	CPU, I/O	table, column	logical	✓	✓	✓	✓	abs. time
Seshadri et al. [97]	centralized	CPU, I/O	table, column	logical	✓	✓	✓	X	rel. time
Learning-based approach									
Ganapathi [60]	centralized	-	runtime info	physical	✓	✓	✓	X	abs. time
Akdere et. al [58]	centralized	-	runtime info	physical	✓	✓	✓	X	abs. time
Wu et al. [32]	centralized	CPU, I/O	runtime info	physical	✓	✓	✓	X	rel. time
Sun et. al [63]	centralized	-	runtime info	physical	✓	✓	✓	X	abs. time

* S — Selection, P — Projection, J — Join, R — Recursive operator, rel. – relative, abs. – absolute

Chapter 4

Cost Estimation for Extended Relational Algebra

For any given query with a set of equivalent plans generated in the query plan space, the goal of the cost model is to select a query plan with the least cost among a set of equivalent query plans. The cost model associates a cost to each relation algebra (RA) operator in a query plan. Cost estimation is a set of mathematical formulas or function which is made up from a complex combination of parameters such as the cardinality, number of distinct attributes, hardware parameters etc, that are used to measure the resources consumption per RA operator or experssion in a query plan and subsequently the whole query plan.

4.1 Introduction

The plan selected by the optimizer in a heuristic-based optimization or planner as the best plan might not always be the cheapest plan. The reason is that operations are not paramtized and nothing guarantees that the selected plan is the best in the pool of query plans generated from the plan space. The same idea goes for cost-based query optimizers as well, if the cost estimation is not achieved adequately, this can lead to the selection of query plans with high resources consumption and large query evaluation time. In the case of recursive queries which are already difficult to optimize, the estimation error might vary significantly from the real cost. The enormous task of choosing the best plan from a pool of equivalent plans requires a cost function that make accurate use of statistical information on data and query to efficiently distinguish the plans based on their cost and estimated behaviour at runtime.

In this chapter we present our contributions on cost estimation techniques for the recursive relational algebraic framework presented in the previous chapter. We reiterate that accurate cardinality estimation is important for efficient query plan cost estimation

and it is one of the building blocks for any cost model. As such, we begin this chapter by presenting in general, the statistics used by our cost estimation technique, the selectivity estimation for tuple-eliminating operators and we discuss the mathematical formula for obtaining the cardinality for each RA operator or expression.

In section Section 4.3, we start by defining our cost functions and assumptions followed by a step-wise cost analysis for each RA operator and present the algorithm and techniques for estimating the maximum number of iterations in a fixpoint in Section 4.5. This to facilitates an accurate cost estimation for fixpoint operator. We then present the cost estimation for fixpoint operator. Improving cardinality estimation has been directly linked to the improvement in the quality of selected query plan [32, 55], as such we integrated state-of-the-art cardinality estimation for RDF graphs [36] into our cost estimation framework.

Finally, we detail how statistics and cost of RA operators are propagated through the query plan tree and give a summary of the chapter.

4.2 Statistical Profile, Selectivity and Cardinality Estimation

As discussed in the previous chapters, a query plan is an ordered sequence of operators organized as a tree; a query tree or plan. The idea of cost estimation is to be able to estimate the cost for each operator in a query tree taking into consideration the dependencies and propagating the cost and statistics through the query tree. In the section we present the set of statistics, and the cardinality and selectivity estimation techniques used by the cost model.

4.2.1 Statistical Profile

The cost model makes use of statistics in order to make an accurate estimation for each operator in a given query tree and for each *QEP* in the plan space. In this section we examine the set of statistics and other parameters that will be used later during the cost estimation.

A base relation refers to a database table. The set of statistics computed for relation variables include the number of rows in the table, the distinct value per column (attribute) and the number of tuples per. These statistics are kept in the system catalogue and used by the cost estimator. This process is automatic in many commercial systems but it can also be updated from time to time as required by simply running the statistics generator. The statistics collected for our use in this work is summarized in Table 4.1.

Data statistics are collected in the database catalogue before the query is run and initially, they are only available for the base relation, intermediate statistics needs to be calculated and propagated through the query tree. The number of tuples and other parameters calculated and propagated through the query evaluation tree.

Table 4.1 – Base relation statistics for a relation E

Parameter	Description
$rowCount(E)$	number of tuples
$relPages(E)$	number of pages that holds the relation E *i.e* $relPages(E) = \frac{rowCount(E)}{b_f}$
b_f	number of tuples that fit into one block: $b_f = \frac{blocksize}{tuplelength}$
$D_{a_1,a_2...,a_n}$	number of distinct values in column $a_1, a_2 \ldots, a_n$
$selFactor$	selectivity
w	weighted factor between *i/o and cpu*
$operCost$	per operator cost

4.2.2 Selectivity Estimation

Selectivity is an important property for calculating the result size of operators that involve the elimination of tuples from a relation such as like filter, join, and antijoin.

Definition 4.2.1. ***Selectivity.*** *Given a relation R whose cardinality is defined as rowCount(R) and a predicate* ***pred*** *which is a boolean function on R. We define the selectivity as the set of tuples in R that satisfy the a predicate* ***pred****.*

$$pred : R \rightarrow \{true, false\}$$

The factor by which the tuples in R reduces after the application of the predicate accepting operator is known as the selectivity factor represented as ***selFactor*** *throughout this thesis.*

$$rowCount(R)^{pred} = rowCount(R) \times selFactor \tag{4.1}$$

$rowCount(R)^{pred}$ *is the number of tuples in R that satisfy the predicate condition*

Sometimes calculating selectivity can be hard. Very popular simplification in many database systems is to apply an empirical constant (e.g. 0.33). So, for any given boolean predicate "$<, \leqslant, \geqslant, >$" on two columns (e.g. column A (pred) Column B), the cost estimator assumes a constant of 0.33. For example the selectivity factor of Column A $<$ Column B is given in Figure 4.1. A relation is said to be more selective if the selectivity factor ($selFactor$) of its predicate is small and less selective if it is high.

The method described above is not accurate enough and could often lead to poor estimation of intermediate result size and should only be considered as the last option. Instead of relying on a constant as described above, the selectivity factor can also be calculated from the distinct attributes per column in the predicate condition. This will, however, require some assumptions on the distribution of data (evenly distributed data

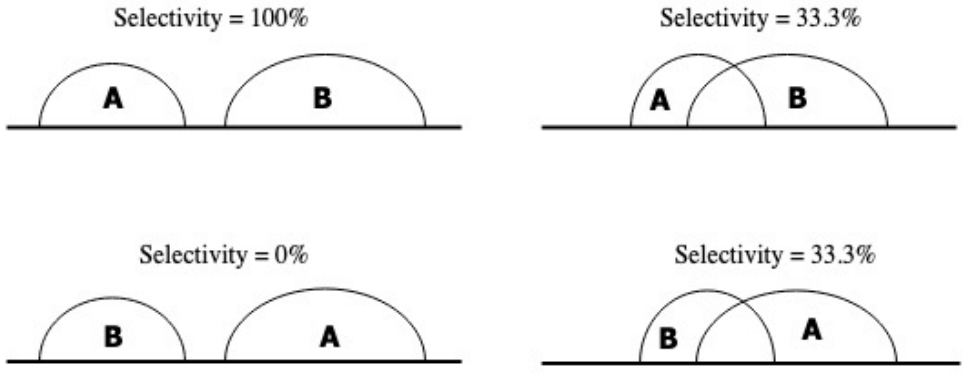


Figure 4.1 – Selectivity factor assuming evenly distributed data (without propagation)

for example) and the cost function needs to properly track the changes in resuls size of a relation in a query tree.

We explore selectivity factor [7] for different predicate conditions of the distribution of attributes in the relations.

- CONDITION 1: $(colum\ A = v)$
  The column (A) selected is the same as the number of attributes present in the relation. This assumes that the values of an attribute are evenly distributed. Values of different attributes are assumed independent of each other.
  If the column is indexed by some index I, the selectivity factor can be given as;

$$selFactor_{(A=v)} = \frac{1}{D_A} \tag{4.2}$$

  where D_A is the estimated number of distinct values in column A.
  If the column is not indexed, the selectivity factor is given as 0.1

- CONDITION 2: $(colum\ A = column\ B)$
  If there are indexes I_1 and I_2 on columns A and B, the selectivity factor can be estimated by;

$$selFactor_{(A=B)} = \frac{1}{max(D_A, D_B)} \tag{4.3}$$

  The formula above assumes that the attribute with smaller distinct values has a corresponding match in the other attribute.

  If only one column has an index I, the estimation can be simplified to;

$$selFactor_{(A=B)} = \frac{1}{D} \tag{4.4}$$

- CONDITION 3: column A *in* v
  When the column selected is contained in the list of the attribute. The selectivity factor for this case is given below;

$$selFactor_{A \in v} = \text{number of attributes in the list} \times selFactor_{(A=v)} \tag{4.5}$$

$$selFactor_{A \in v} = \text{number of attributes in the list} \times \frac{1}{D} \tag{4.6}$$

- CONDITION 4: Range selections $(A > v)$
  The statistics about the maximum (D_{high}) and minimum (D_{low}) distinct values of the relation is obtained, the selectivity is given as follows;

$$selFactor_{(A>v)} = \frac{D_{high} - v}{D_{high} - D_{low}} \tag{4.7}$$

  D_{high} represents the high key value and D_{low} is the low key value.

- CONDITION 5: NOT pred
  The selectivity factor for this case is given as;

$$selFactor_{(\text{NOT pred})} = 1 - selFactor(pred) \tag{4.8}$$

  This case is considered a less selective case and the selectivity factor can also be estimated as 0.9.

4.2.2.1 Histograms

The traditional selectivity estimation method described above rely on several assumptions, one assumption is that efficient intermediate result size and distinct attributes estimation, and uniform attribute distribution are already calculated. The reality is that estimating intermediate result size is difficult as data are not often uniformly distributed in real datasets.

Histograms are convenient technique to capture data distribution and they are used to store frequency of attributes of a relation. Histograms are particularly useful for determining the selectivities of predicates and subsequently the result size of queries. Basically, data is partitioned into buckets representing intervals where values are captured within these intervals. Each bucket is equiped with the *most common value (mcv)* and the count of these attributes in each bucket is referred to as the *most frequent value (mfv)*. These information about the histogram values and frequencies are stored in the database catalogue.

Some popular types of histigram include equi-width and equi-depth histograms. The mfv of each buckets are different in equi-width histogram while the bucket length of each bucket is the same. For equi-height histogram, the mfv of each bucket is the same but the interval length may vary.

4.2.3 Cardinality Estimation for Non-recursive Terms

Cardinality estimation is an important ingredient in query optimization and it has been a subject of many research works [10, 32, 36, 38, 40, 41, 55, 56, 98] and many others. [55] reported that cardinality estimation plays a central role in the quality of plan selection in query engines.

In a typical query tree, cardinality of the base relation is propagated through the query tree as different expressions or operations are applied. Most of the operators described in the previous chapter usually appear as an intermediate node or level in a query plan tree. As a result we need to be able to somehow account for the changes introduced to the intermediate relations as a result of previous evaluation or computation performed on them.

Definition 4.2.2. ***(Cardinality of a term)*** *Given a term* φ *expressed as* $G = (V, E)$. *Let* $v_1, ...v_n \in V$ *a finite set of nodes, we say;*

- *rowCount(V) is the cardinality of* G *which also holds for* φ
- $\texttt{distinct}(V)_c$ *is the number of distinct nodes*

A relation is accessed through a relation variable or constant mapping. Estimating the cardinality for a base relation is straightforward but what is more challenging is estimating these cardinalities for the other operators in the query tree, for example filter and join operations might affect the number of rows in the relation.

While estimated cardinality might not always represent the true number of tuples in the intermediate relations, they approximate the proportion of changes in the dataset.

Table 4.2 – Cardinality Estimate for μ-RA operators

#	**Operator**	**Cardinality**
Let $c = cost(\varphi)$, *cost* is given in Section 4.3.1.		
1.	E	$rowCount(E)$
2.	X	1
3.	$\lvert c \rightarrow v \rvert$	1
4.	$\rho_a^b(\varphi)$	$c.rowCount$
5.	$\pi_{a_1}(\varphi)$	$c.rowCount \times rFactor$
6.	$\sigma_{a_1}(\varphi)$	$c.rowCount \times selFactor$
Let $c_1 = cost(\varphi_1)$, $c_2 = cost(\varphi_2)$ and $selFactor$ is the selectivity factor		
7.	$\varphi_1 \cup \varphi_2$	$c_1.rowCount + c_2.rowCount$
8.	$\varphi_1 \bowtie \varphi_2$	$c_1.rowCount \times c_2.rowCount \times selFactor$
9.	$\varphi_1 \triangleright \varphi_2$	$c_1.rowCount \times c_2.rowCount \times selFactor$

In Table 4.2, we present the cardinality estimation formulas for the extended relation algebra operators discussed in the previous section. In any part of this work we interchangeably refer to the cardinality of operators as the number of tuples, $rowCount$, and result size. We discuss in details how we obtained the cardinality formula for each operator in Section 4.4.

4.3 Cost Estimation for Extended RA

In this section we present the definition of cost model component used in this work and present the cost function.

4.3.1 Definitions

We can define a cost model as a (set of) numerical functions that accepts as input some parameters like relation size, information about index and access methods, selectivity estimate etc. and output a numerical or set of numerical values. Cost-based optimization is the process of obtaining the plan with the minimum cost using the estimated cost values calculated from input parameters. Following [99], we define a cost model and it components as follows;

Definition 4.3.1. ***(Cost Model).*** *A cost model can be defined as;*

$$\mathrm{costModel} = \langle \beta,\ \mathbb{S}_i,\ \mathrm{F}n,\ \delta \rangle$$

where;

- $\beta \subseteq \{\mathrm{CPU}, \mathrm{IO}\}$ is the cost component type which effect the factors that made up the cost estimation. In our case, the cost can be expressed as a summation of all these CPU and I/O parameters.
- $\mathbb{S}_i$ a set of resources or parameters assigned to the cost component type
- $\mathrm{F}n$ is the cost function
- $\delta \subseteq \{\text{abs. time},\ \text{rel. time},\ size\}$ is the objective function of the cost model

Definition 4.3.2. ***(Cost Parameters).*** $\mathbb{S}_i = \{\mathrm{res}_1,\ \mathrm{res}_2, \mathrm{res}_3 \ldots\ \mathrm{res}_n\}$ *is a set of parameters such that* $\mathrm{res}_i \in \mathrm{Res} = \{\mathrm{db}_{params},\ stats,\ \mathrm{qep},\ selFactor,\ \tau\}$

where;

- db_{params} is the database and hardware parameters
- $stats$ is the set of statistics maintained on the data
- selFactor is the selectivity factor
- τ are other constants such as the weighted CPU and I/O constant

Definition 4.3.3. ***(Objective Function).*** *The measure of the cost function is referred to as the objective function (δ). $\delta \subseteq \{abs.\ time, rel.\ time, size\}$*

where;

- abs. time refers to the computation in absolute time
- rel. time refers to the computation in relative time
- size is the measure of query function in terms of the number of rows or block size returned in the result

Definition 4.3.4. ***(Cost Function)*** *Cost calculation is achieved through a function Fn that is based on a cost component type β which uses a set of parameters or resources $\mathbb{S}_i$ and output the cost in terms of δ.*

$$Fn = \langle s,\ \delta \rangle \qquad where\ s \subseteq \mathbb{S}_i$$

Definition 4.3.5. ***(Cost Minimization)*** *Given a set of alternative query plan $QEP_{alt} = \{P_1, P_2 \ldots, P_i\}$, a set of resources $\mathbb{S}_i$, the cost optimization function selects a plan that minimizes the cost function such that;*

$$P^* = \underset{P \in QEP_{alt}}{\arg\min}\ Fn(P_i, \mathbb{S}_i) \tag{4.9}$$

4.3.2 Assumptions

In order for our cost model to compute the cost for a query plan, certain assumptions are made. These assumptions include;

- Tuples are stored in blocks
- There are several tuples stored in a block
- We assume that only sequential scan is possible since we have no information on the index
- Each operator in the query tree carries a cost and the total cost of a query plan is the summation of all these individual costs.

4.3.3 Cost Function

We propose a cost estimation technique suitable for algebraic framework (terms) of the extended relational algebra [26] we follow the initial idea first described in the seminal approach of System R [7] and followed by extensive works on the topic [10, 30, 32, 55, 61]. The cost function computes an estimated cost for every query plan in the plan space. Following our previous defintion of a cost fucntion F_n, our cost estimation function accepts a query plan, data statistics and returns a measure in terms of the relative time i.e. $evalCost$ and $rowCount$. For notational conveniency, we refer to the cost function F_n as "cost" throughout the rest of this work.

Based on the aforementioned syntax in Chapter 3, we define a cost estimation function for a term φ as:

$$cost(\varphi) = (evalCost, rowCount) \tag{4.10}$$

where;

- $evalCost$ is the estimated computation cost and,
- $rowCount$ is the estimated result size, i.e. the number of tuples returned

$cost(\varphi)$ keeps track of the *evalCost and rowCount*. The function $cost(\varphi)$ is paramatized by the statistics of the input relation and is defined recursively using a bottom up approach, starting from the tree leaves (constants and relational variables).

The *evalCost* follows a general formula given by [7] as given in Equation 4.11

$$evalCost = CPU_{cost} + I/O_{cost} * w \tag{4.11}$$

where;

- CPU_{cost} is the cost of executing *cpu instruction*,
- IO_{cost} is the *disk I/O* cost times a weighted average between the *cpu and io*.
- w is the weighted factor between the CPU and I/O

4.4 Cost Analysis for Non-recursive RA Operators

In this section, we present the cost formulas for the non-recursive relational algebraic operators described in previous chapters. Query processing starts by accessing records from the relation or table through relation variable or through constant record, as such we derive the cost formula for the different operators starting from these base cases — relation variable and constant and then define the cost for the other operators.

4.4.1 Relation Variable

The relational variable denoted by E is the fundamental variable that retrieve the relational data stored on disk and it is from this that the cost of every node or operator is computed. Given a base relation E, the cost of a *relation variable* $\mathtt{cost}(E)$ is calculated as follows;

Recall Equation 4.11,

$$evalCost = \mathtt{relPages}(E) + \mathtt{rowCount}(E) \times w \tag{4.12}$$

rowCount(E) and relPages(E) are the ones retrieved from the base relation. w is the weighted factor

$$\mathtt{cost}(E) = \big(evalCost, \mathtt{rowCount}(E)\big) \tag{4.13}$$

The input of other operators in a query tree can be sub-expression computed directly from a relation variable or indirectly through proper estimation of intermediate result size and cost.

4.4.2 Recursive Variables

A *recursive variable (X)* returns the unit result size estimate of the variable supplied to it. Therefore, the $\mathtt{rowCount}$ and cost are both 1.

$$\mathtt{cost}(X) = (1.0, 1.0) \tag{4.14}$$

4.4.3 Constant Mapping

Constant mapping ($|c \to v|$) is a one-tuple-one-attribute relation. The cost is given as;

$$\mathtt{cost}(|c \to v|) = (1.0, 1.0) \tag{4.15}$$

4.4.4 Filter

Filter involves selecting only tuples that satisfy a given predicate condition. Written as $\sigma_f(\varphi)$, *Filter* involves traversing the rows of a relation and comparing them with the filter condition. This is because the result returned consists only of tuples that satisfy the predicate thereby reducing the result size.

There are two $\mathtt{filter}$ conditions available in μ-RA [26];

- $\mathtt{True}()$: this is the default condition which means that all tuples satisfy the predicate and all of the tuples will be returned as the result size.

  Let $c = \mathtt{cost}(\varphi)$, the cost and result size in this case is given as the cost of φ term.

  $$\mathtt{cost} = (c.evalCost, c.\mathtt{rowCount}) \tag{4.16}$$

- $\mathtt{Equal}\ (\mathtt{col} = value)$: where $\mathtt{col}$ is the specific column to *filter* and $value$ is used to exclude tuples that do not satisfy the predicate condition. In this case, the number

of tuples returned in this case is expected to be lesser than the original number of tuples. The cost of applying the filter operation on a term φ written as $\sigma_f(\varphi)$ can be expressed as follows;

Let $c = Cost(\varphi)$ be the cost of φ, where $c.evalCost$ = cost of scanning the relation and $c.rowCount$ = number of tuples in the relation.

The number of *tuples* returned after a *Filter* operation (*i.e. rowCount*) is applied on a relation is given as;

$$rowCount = \text{Total \#tuples in the relation} \times selFactor \tag{4.17}$$

where $selFactor$ is the selectivity of the applicable predicate. The *result size of Filter* can be calculated as;

$$rowCount = c.rowCount \times selFactor \tag{4.18}$$

The *evaluation cost* is given as follows;

$$evalCost = c.evalCost + \text{Cost of tuple elimination} \tag{4.19}$$

Also,

$$\text{Cost of tuple elimination} = \text{\#tuples left after filter has been applied} \tag{4.20}$$

And the evaluation cost is also given below;

$$evalCost = c.evalCost + c.rowCount \tag{4.21}$$

4.4.5 Anti-projection

Anti-projection written as $\tilde{\pi}_A(\varphi)$, removes only attributes of the relation mentioned in A. *Anti-projection* here is the opposite of *SQL projection* which returns only attributes specified in the operator. This operator is set-based and removes duplicates from the original relation. Lets consider the following example;

As demonstrated in the Table 4.3,

- the result is constructed in the relation E by discarding the attributes on d_1, d_2, returning the remaining attributes d_3, d_4, d_5 of the relation as the result.
- we argue that the more columns we remove from the relation, the more the probability of having tuples discarded from the relation.

Table 4.3 – Anti-Projection Example

d_1	d_2	d_3	d_4	d_5
1	1		1	1
2	2	2		2
3	3	4		
4		3	4	
5	5	5		5
6				
7	7			

A relation E
before anti-projection

$\tilde{\pi}_{(d_1,d_2)}(E)$

d_3	d_4	d_5
1		1
2	2	2
3	3	
	4	
5		5

Relation E
after removing columns (d_1, d_2)

We introduce the notion of *reduction factor* (rFactor). The *rFactor* is calculated by taking the ratio of the number of attributes to be removed and the total number of attributes in the relation.

$$\text{rFactor} = 1 - \frac{\#\text{attr. in projection}}{\text{total attr. in relation}} \tag{4.22}$$

Unlike *Filter*, this operator does not traverse the rows of a relation. The cost of *anti-projection* operation on a term φ written as $\tilde{\pi}_A(\varphi)$ can be expressed as follows;

Let $c = \text{Cost}(\varphi)$ be the cost of φ, where c.*evalCost* = cost of scanning the relation and c.*rowCount* = number of tuples in the relation

The number of tuples left after an *anti-projection* operation is estimated as follows;

$$\text{rowCount} = \#\text{Rows in the relation} \times \text{rFactor}$$

$$\text{rowCount} = c.\text{rowCount} \times \text{rFactor} \tag{4.23}$$

The evaluation cost *evalCost* for *anti-projection* is estimated as the cost of φ.

$$\text{evalCost} = \textit{cost of scanning the relation} \tag{4.24}$$

$$\text{evalCost} = c.\text{evalCost} \tag{4.25}$$

4.4.6 Union

Given two relations R and E, the union ($\cup$) operation corresponds to the compatible set of tuples from R and E. The union of two terms R and E is written as $R \cup E$.

The example below retrieves the list of all countries of employees and customers.

We can see that the duplicates (*France, Japan*) were removed from the result.

The following steps describe the processes involved in the cost estimation of a *union* operator.

- For relation R
  - Read blocks of relation *R*
  - For each tuples $t_{country}$ of relation *R*, output $t_{country}$

Table 4.4 – Union example

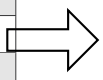
R

emp_id	country
1	Japan
2	France
3	Australia
4	Germany

$R_{country} \cup S_{country}$

S

cust_id	country
1	UK
3	France
4	Nigeria
5	Japan



Result

country
Japan
France
Australia
Germany
UK
Nigeria

- For relation S
  - Read blocks of relation S
  - For each tuples $t_{country}$ of relation S, output $r_{country}$
- Remove duplicates from result

Given two terms φ_1 and φ_2, the cost estimate for *union* between these two terms can be estimated as follows;

Let $c_1 = cost(\varphi_1)$ and $c_2 = cost(\varphi_2)$, the cost for terms φ_1 and φ_2 respectively.

The result size (*rowCount*) of *union* operator can also be estimated as follows;

$$rowCount = \#\text{tuples in the left rel.} + \#\text{tuples in the right rel.} \tag{4.26}$$

The result size, $rowCount$ given in Eq. 4.27 below is an overestimation for the result size of the union because duplicates are eliminated from the final result.

$$rowCount = c_1.rowCount + c_2.rowCount \tag{4.27}$$

The evaluation cost (*evalCost*) of a *union* operator is given as;

$$evalCost = \text{cost of scanning the left rel.} + \text{cost of scanning the right rel.} \tag{4.28}$$

The computation cost, $evalCost$ for $union$ on φ_1 and φ_2 is computed from the cost of each term. *We assume that the cost of duplicate elimination is negligible since it can be performed in O(n) time.*

$$evalCost = c_1.evalCost + c_2.evalCost \tag{4.29}$$

4.4.7 Join

Given two relations R and S, *join* operator involves combining compatible pairs of tuples from R and S. The two relations must be $join$ compatible. The $join$ $(\bowtie)$ of two terns R and S is written as $R \bowtie S$.

To illustrate *join*, we consider the two relations below;

Joining the two tables above (*R and S*) on the *user_id and id* gives the *user name* and their *likes*.

Table 4.5 – Join example

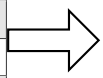
R

id	name
1	Yves
2	Jones
3	Sam
4	Larry
5	Bode

$R \bowtie_{id=user\_id} S$

S

user_id	likes
3	Soccer
1	Racing
1	Movies
4	Coding
6	Kenny



Result

name	likes
Sam	Soccer
Yves	Racing
Yves	Movies
Larry	Coding

Given two terms φ_1 and φ_2, the cost estimate for join between these two terms can be estimated as follows;

Let $c_1 = cost(\varphi_1)$ and $c_2 = cost(\varphi_2)$, the cost for terms φ_1 and φ_2; the small and big relations respectively.
where;

- $c_1.evalCost$ = cost of scanning the small relation
- $c_1.rowCount$ = number of tuples in the small relation
- $c_2.evalCost$ = cost of scanning the big relation
- $c_2.rowCount$ = number of tuples in the big relation

The result size (*rowCount*) of join operator can also be estimated as follows;

$$rowCount = \#\text{tuples in the left rel.} \times \#\text{tuples in the right rel.} \times Selectivity \tag{4.30}$$

The selectivity estimation mostly rely on the distinct values per attribute of the relation. But because this distinct values might have changed (*e.g* reduced) before a join operation is performed on the relations (or intermediate relations), for example, performing a filter on one of the join tables before applying a join will significantly affect the cardinality of the result size. To this end, we adopt the work of [35] for calculating the *number of distinct values* of the join column which will later be used for selectivity estimation.

We then calculate the *join size*

$$rowCount = c_1.rowCount \times c_2.rowCount \times selFactor \tag{4.31}$$

For the evaluation cost *evalCost*,

$$\begin{aligned} evalCost = {} & \text{Cost of scanning the small rel.} + \text{Cost of finding matching tuples} \\ & + \text{Cost of gathering the results} \end{aligned} \tag{4.32}$$

Cost of finding matching tuples corresponds to the cost of scanning the contiguous group of the bigger relation which corresponds to one join column value in the smaller relation.

$$\begin{aligned}\text{Cost of finding matching tuples} = \text{cost of scanning the big relation} \\ \times \text{number of tuples in small rel.}\end{aligned} \tag{4.33}$$

The cost of finding matching tuples in the big relation can therefore be given as;

$$\text{Cost of finding matching tuples} = c_1.rowCount \times c_2.evalCost \tag{4.34}$$

The *Cost of gathering the results* is estimated as the number of tuples obtained after the join (*i.e. join size*) given below;

$$\text{Cost of gathering the results} = \text{join size} \tag{4.35}$$

The *cost of scanning the big relation* is already given above as $c_2.evalCost$ and *join size* is $rowCount$ given in Equation 4.31 above.

Evaluation cost of *join* can be estimated as follows;

$$evalCost = c_1.evalCost + (c_1.rowCount \times c_2.evalCost) + \text{join size} \tag{4.36}$$

One of the key strategies we adopt in estimating the cost of *join* is ensuring that the left relation is the smallest of the two relations.

4.4.8 Anti-join

Given two relations R and S, *anti-join* ($R \triangleright S$) returns the mappings from R that has no matching mapping in S. The tuples retained from the relation after performing anti-join are those ones from R with no match in S.

With *anti-join*, the inner relation R is first evaluated and the result is used as a predicate for the outer relation S.

Given two terms φ_1 and φ_2, the cost for anti-join can be estimated as follows;

Let $c_1 = cost(\varphi_1)$ and $c_2 = cost(\varphi_2)$, the cost for terms φ_1 and φ_2 respectively. where;

- $c_1.evalCost$ = cost of scanning the outer relation
- $c_1.rowCount$ = number of tuples in the outer relation
- $c_2.evalCost$ = cost of scanning the inner relation
- $c_2.rowCount$ = number of tuples in the inner relation

We then calculate the *anti-join size*

$$rowCount = c_1.rowCount \times selFactor_{Not(\varphi_1 = \varphi_2)} \tag{4.37}$$

Here $selFactor$ is the cardinality of the outer relation (φ_1) multiplied by the selectivty factor of "NOT $pred$" (see section 4.2.2). "$pred$" here indicates that the selectivity factor for "$column\ A = column\ B$".

The selectivity $selFactor_{Not(a=b)}$ is given by the equation below;

$$selFactor_{Not(\varphi_1=\varphi_2)} = 1 - selFactor_{(col\ a\ =\ col\ b)} \tag{4.38}$$

This selectivity is retrieved from the histogram values of the database catalogue. In the absence of this histogram selectivity value, we use the number of distinct values D_1, D_2 for the participating columns to estimate the selectivity factor.

$$selFactor_{Not(\varphi_1=\varphi_2)} = 1 - \frac{1}{max(D_1, D_2)} \tag{4.39}$$

The computation cost $evalCost$ for anti-join on φ_1 and φ_2 is computed from the cost of each term as given in Equation 4.40.

$$evalCost = c_1.evalCost\ +\ (c_1.rowCount \times c_2.evalCost)\ +\ c_1.rowCount \tag{4.40}$$

With anti-join, the inner relation φ_2 is first evaluated and the result is used as a predicate for the outer relation φ_1.

4.5 Fixpoint Operator

We re-introduce the fixpoint operator presented in Chapter 3 here and discuss the challenges encountered in estimating its cost. A cost analysis of fixpoint operator is conducted and we present our technique for estimating the maximum nnumber of recursive steps in a recursive operator, the cardinality and its cost estimates.

A fixpoint operator $f(s)$ iteratively evaluates the function f on s until a certain predicate is satisfied [4, 22, 54]. Basically, the function f is iteratively applied on s and reaches the $fixpoint$ at *nth*-iteration – the point at which the operation terminates. At the n^{th} iteration, $f^n = f^{n+1}(s)$.

$$\mu X.\ \underbrace{\varphi}_{\text{constant part}}\ \cup\ \underbrace{(R \bowtie \varphi)}_{\text{recursive part}} \tag{4.41}$$

Recursive terms contains two parts: the *constant part* and the *recursive part* as depicted in Equation 4.41. The *recursive part* is executed several times until it no longer produces further results. The *constant part*, executed just once, provides the initial results used as a starting point for the recursion (see [26] for a formal semantics).

4.5.1 Problem Definition

A challenging part consists in determining when the recursive part of the fixpoint operation terminates. This is crucial for estimating the number of rows returned and subsequently the cost of the whole fixpoint operator.

Definition 4.5.1. ***Maximum number of iteration.*** *The number of iterative steps* $\mathbb{N}$ *is the total path from node* s *to* t *consisting of one or more adjacent edges that are mutually reachable from* s*.*

$$(s, x_1),\ (x_1, x_2),\ (x_2, x_3), \ldots\ (x_{n-1}, t)$$

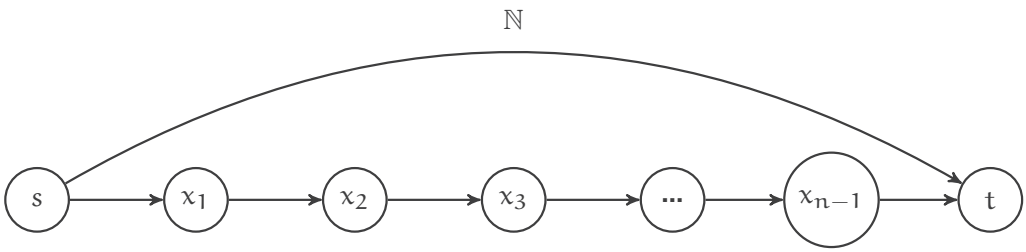


Figure 4.2 – Maximum iteration in a recursive path traversal

In Figure 4.2, a node s is connected to a node t through paths x_1 to x_{n-1}, these edges could also be connected to other nodes but for the sake of simplicity we assumed a simple scenario. Depending on the connection between nodes in a graph or the structure of the graph, representing the maximum number of iterative steps for the node traversal as a constant could significantly lead to cost bad estimation and the selection of low quality query plan.

Another problem that could emanate from this is that the performance gain of queries observed through algebraic transformation rules like predicate pushdown can be missed. Predicate pushdown is a tuple-eliminating algebraic rule where operators like filter is pushed as close to the source as possible in a query tree enabling a decrease in the result size of intermediate expressions thereby reducing the cost and execution time. Bad cost estimation approximation that originate from using a constant number of iteration for recursive query cost estimation can lead the optimizer to missing such transformation rules that are beneficial for overall query performance.

Let us consider an example, Figure 4.3 represents the query plans for our Covid-19 example in Chapter 1. There are two different plans represented on the tree. In Plan 1, after computing the fixpoint, the result returned is 5000 rows. The number of people that have been in contact with someone is 1000, we then combine the result of the fixpoint with the list of people that have been in contact with someone using a join operator which results in 1500 rows containing people who have been in contact with any person that was with a Covid-19 patient.In the final step, we filter by join i.e. we only select people that John has been in contact with and a connected list of people those people have been with. The total number of tuples returned for the query is 600.

Contrarily, in Plan 2 all the computation is done inside the fixpoint. We start by finding the a list a people that John has been in contact with in the constant part of the fixpoint

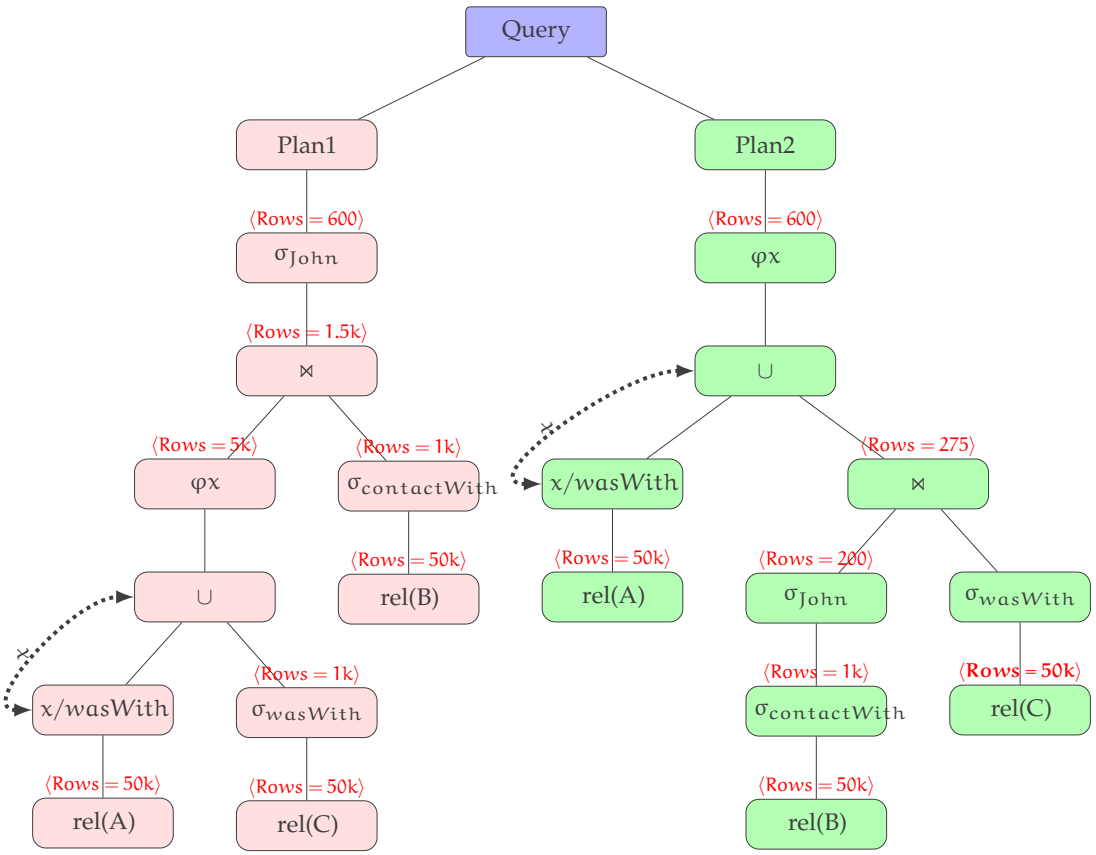


Figure 4.3 – *QEP* for Covid-19 contact tracing example in Chapter 1

which are 200 rows in total, we then join that with the list of people he has been with. The intermediate result at this point (after the join) yields 275 rows. The result is combined with the final result from the recursive part of the fixpoint as a union and the result (600 rows) is returned.

For better understanding, the difference between Plan 1 and Plan 2 in Figure 4.3 is that because the filter (σ_{JOHN}) is pushed down inside the fixpoint in Plan 2 the number of intermediate result participating in the join is reduced, enhancing query evaluation time. The time spend performing the join differs in the two queries due to this. The disparity in query time as a result of this algebraic transformation rule and cost-based plan selection can range from a few milliseconds to minutes and even days.

4.5.2 Iterative Step Analysis

In the previous section we have already emphasized the importance of estimating the maximum iterative steps rather than using constant like being used in many data processing systems that support recursion (linear or non-linear). We describe the process at each recursive step followed by an algorithms for estimating the number of iterative steps in the recursive part of a fixpoint operator.

To estimate the cost of a fixpoint, we follow the steps below:

Table 4.6 – Recursive steps computation in the fixpoint operator

Step	Input	Computation	Step termination
0	X	$\varnothing$	–
By iterative susbstitution of X in Section 6.3.2, where $selFactor$ = selectivity			
1	$\varnothing$	$E \cup (R \bowtie \varnothing) = E \Rightarrow X_1$ $\Delta X_1 = X_1 \mid \Delta X_1 \neq \varnothing$	$\Gamma_1 = rowCount(E)$
2	X_1	$X_1 \cup R \bowtie \Delta X_1 \Rightarrow X_2$ $\Delta X_2 = X_2 - X_1 \mid \Delta X_2 \neq \varnothing$	$\Gamma_2 = \Gamma_1 * rowCount(X_1 \cup R) * selFactor$
3	X_2	$X_2 \cup R \bowtie \Delta X_2 \Rightarrow X_3$ $\Delta X_3 = X_3 - X_2 \mid \Delta X_3 \neq \varnothing$	$\Gamma_3 = \Gamma_2 * rowCount(X_2 \cup R) * selFactor$
...	...	...	...
n	X_{n-1}	$X_{n-1} \cup R \bowtie \Delta X_{n-1} \Rightarrow X_n$ $\Delta X_n = X_n - X_{n-1} \mid \Delta X_n = \varnothing$	$\Gamma_n = \Gamma_{n-1} * rowCount(X_{n-1} \cup R) * selFactor$ $\Gamma_n \leqslant selFactor \wedge \Delta X_n = \varnothing$

1. we start from X which is initially an empty relation ($\emptyset$), we substitute X into the equation of the fixpoint and perform a union, the whole fixpoint term then reduces to $\varphi \cup (R \bowtie \emptyset) = \varphi$, thus we have $rowCount = rowCount(\varphi)$

2. at this step, the value of X is now $\varphi \cup (R \bowtie \varphi)$. Given the cardinality of φ and R and the selectivity factor of the join, by substituting the result of X (i.e $R \bowtie \varphi$), we compute the $evalcost$ and $rowcount$ of this step;

3. by iterative substitution of X, the computation continues repeatedly until some step N such that the result size is less or equal than the initial selectivity factor, i.e. $rowCount \leqslant Sel$. At this point, we estimate that the maximum number of iterations has been reached and that the iteration terminates.

The steps are described in Table 4.6

Theorem 4.5.1. ***(Fixpoint Convergence)*** *If f is a recursive function applied on x, then*

$$f(x),\ f_1(x),\ f_2(x),\ \ldots\ f_i(x)$$

Thus the iteration continues until

$$\Delta f_n(x) = f_n(x) - f_{n-1}(x) = \varnothing, \quad \textit{for some } n > 0$$

In algorithm 4, we present the algorithm for computing the result size of *fixpoint* operation.

The step and result size estimation relies on several assumptions, that are inspired by the so-called *semi-naïve* evaluation of transitive closures found in the literature [4, 17, 21, 24]. In particular, we assume that only the new results generated by an iteration are used for the next iteration and that the number of tuples reduces until a maximum number

Algorithm 4 : Algorithm for computing #steps and size of recursive part

```
1: function RECURSION(rowCount(φ), rowCount(R ⋈ φ))
2:     X ← ∅
3:     step(X) ← 1
4:     size ← rowCount(R ⋈ φ)                          ▷ Calculate join size
5:     while size ⩾ selFactor do
6:         size ← size ∗ selFactor
7:         step(X) ← step(X) + 1
8:     return step(X)                  ▷ The maximum #step of the recursive part
```

of iterations $\mathbb{N}$ is reached. At each step, the result size is reduced by a factor s which we compute from the base case of the *fixpoint (i.e.* $R \bowtie \varphi$*)*. We estimate the number $\mathbb{N}$ of iterations as:

$$\mathbb{N} = \log_s(K) \tag{4.42}$$

where $K = \text{rowCount}(R \bowtie \varphi)$ is the estimated number of tuples in the recursive part of the fixpoint.

4.5.3 Cardinality Estimation

To estimate the cardinality of a fixpoint operator we apply the knowledge of the maximum number of iterative steps and the selectivity factor. The result size of the constant part of the fixpoint computation is fairly deterministic and can be estimated from the intermediate result size of its children node operators or sub-expressions.

As stated above, in each iterative step, the result size increases until a maximum iteration n is reached. We noted that the result size grow by a factor selFactor. In a decomposed fixpoint (consisting of constant and recursive part), the recursive computation requires at least $k-1$ joins of $R \bowtie X$ (where X is recursive). In the first step of the iteration $X = E$ and the first join is $R \bowtie E$.

> **Theorem 4.5.2.** ***(Step-wise Cardinality)*** *If the result size grow by a factor* selFactor *and the total iterative steps is* $\mathbb{N}$*, then we can say that the total number of tuples after the recursion is;*
>
> $$\text{rowCount}(R \bowtie \varphi)^{\mathbb{N}} = c_1.\text{rowCount} * (1 + \text{selFactor})^{\mathbb{N}} \tag{4.43}$$
>
> *where* $c_1.\text{rowCount}$ *is the result size of the join* $(\text{rowCount}(R \bowtie \varphi)^{\mathbb{N}})$ *at step one.*

Table 4.7 shows the cardinality estimation formula at each iterative step.

Figure 4.4 shows the graph of increment of the result size of the recursive computation in a fixpoint using Equation 4.43. The cardinality keeps growing until it reachs a point; the maximum $\mathbb{N}$ where no new results are available, then the computation terminates and the result is gathered and merged with the intermediate results from the constant part of the fixpoint computation.

Table 4.7 – Cardinality estimation for recursive computation

Step	Formula
$n = 0$	$c_1.rowCount$
$n = 1$	$c_1.rowCount * (1 + selFactor)$
$n = 2$	$c_1.rowCount * (1 + selFactor)^2$
$n = 3$	$c_1.rowCount * (1 + selFactor)^3$
...	...
$\mathbb{N}$	$c_1.rowCount * (1 + selFactor)^{\mathbb{N}}$

* $selFactor = 0.05$

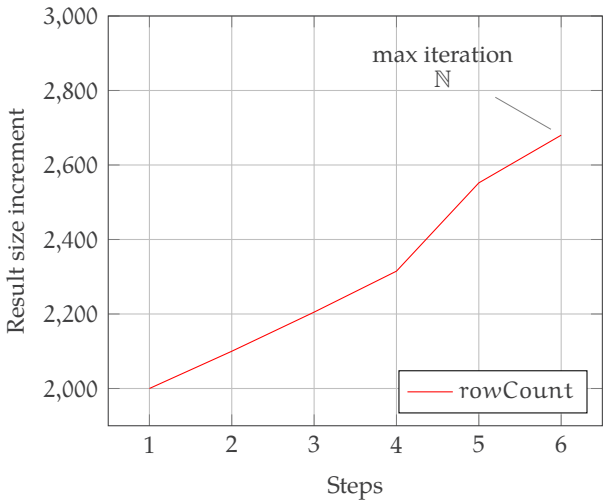


Figure 4.4 – Result size growth through step $\mathbb{N}$

4.5.4 Cost Estimation for Fixpoint Operator

We extend the notion of our cost model (discussed earlier) to recursive queries over graph database by presenting the cost formula for a fixpoint operator. We now know how to compute the number $\mathbb{N}$ of iterative steps. We proceed to compute the cost of the overall fixpoint term. Let $c_1 = cost(\varphi)$, $c_2 = cost(R)$ respectively:

- $c_1.evalCost$ = cost of computing φ
- $c_1.rowCount$ = number of tuples in φ
- $c_2.evalCost$ = cost of computing the recursive relation R
- $c_2.rowCount$ = number of tuples in the recursive relation R

The evaluation cost for a *fixpoint* is given as;

$$evalCost = c_1.evalCost + (c_2.evalCost \times \mathbb{N}) + rowCount \tag{4.44}$$

At *step 1* above, X is empty, then the *rowCount* at step 1 is;

$$rowCount(X) = c_1.rowCount \tag{4.45}$$

The evaluation cost at this point is estimated as $c_1.evalCost$ and $c_2.rowCount$ respectively. We then estimate the join size *i.e.* $rowCount(R \bowtie \varphi)$

$$rowCount(R \bowtie \varphi) = c_2.rowCount \times c_1.rowCount \times selFactor \tag{4.46}$$

The evaluation cost is estimated as:

$$\begin{aligned} evalCost = {} & \text{cost of computing const. part} + (\mathbb{N} \times \text{cost of scanning rec. part}) \\ & + \text{cost of gathering the results} \end{aligned} \tag{4.47}$$

We estimate the *cost of gathering the results* $Cost_{res}$ as the maximum of the cardinality of relation E (i.e. $c_1.rowCount$) and $rowCount(R \bowtie \varphi)$.

$$Cost_{res} = max(c_1.rowCount, rowCount(R \bowtie \varphi)) \tag{4.48}$$

The result size (*rowCount*) of a fixpoint operator is estimated as;

$$rowCount = c_1.rowCount + rowCount(R \bowtie \varphi))^{\mathbb{N}} \tag{4.49}$$

where $rowCount(R \bowtie \varphi))^{\mathbb{N}}$ is the result size after the computation of recursive part of the fixpoint operator at step $\mathbb{N}$.

4.6 Improving Cardinality Estimation with Graph Summarization

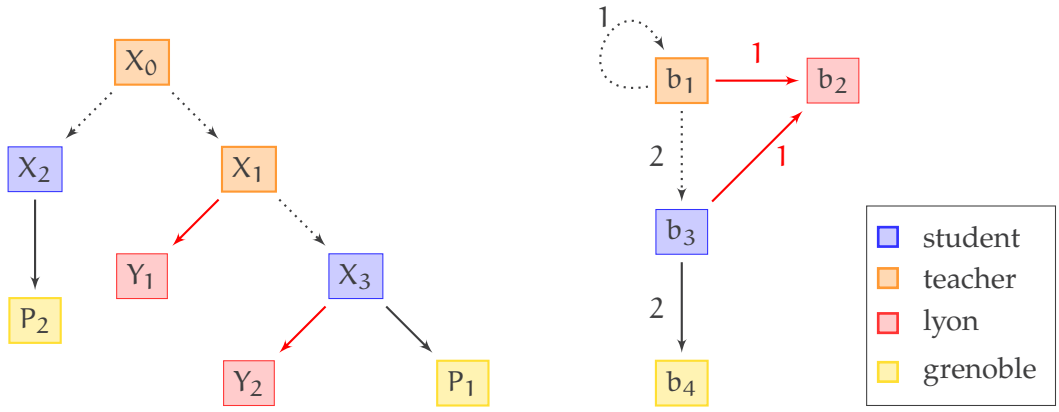
RDF graph summarization is the process of compressing graph by merging their nodes and extracting meaningful summaries from RDF knowledge depicting as close as possible the actual contents of the graph [36, 100]. Categorization of studies on RDF graph summarization has been conducted in [100] and these categories include *(i)* structural methods consider both paths and subgraphs in RDF graph, *(ii)* pattern mining methods are based on data mining techniques *(iii)* statistical methods involves quantitative summarization of graph using counting instances or histograms and *(iv)* hybrid methods combining the different approaches. We refer our readers to [100] on categorization of graph summarization techniques. The different categories of graph summarization techniques target different applications.

4.6.1 SumRDF

In this work we are interested in the application of graph summarization to cardinality estimation. We adopt the work of [36] (SumRDF), a hybrid RDF graph summarization

technique used for estimating the cardinality for RDF queries of the basic graph pattern. SumRDF [36] cardinality estimation technique for RDF graphs is based on grouping nodes that have exactly the same set of types, outgoing and incoming properties.

Definition 4.6.1. ***(Summary)*** *A Summary* $S = \langle H, w, \mu \rangle$ *is a triple where* H *is an RDF graph called the summariation graph,* $w : H \to \mathbb{N}$ *is a weight function assigning a positive natural number to each triple in H, and* α *is a surjective summarization function from a finite set of resources* $\mathrm{dom}(\mu)$ *to resources of* H $\mathrm{res}(H)$*. The resources of* $\mathrm{res}(H)$ *are called buckets.*



a RDF graph G b Summary S of graph G c Legend

Figure 4.5 – Graph summary example

Let α be the summarization function consisting of a many-to-one mapping from a data node n to a summary bucket b. Nodes are grouped into buckets based on their types and any two nodes that have the same type are collapsed into the same bucket. As summary size may be large for instance for very large graphs, the the target size has been introduced. A target size (specified by the user) can be used to reduce the growing size of the summary and it works by merging nodes with similar incoming and outgoing properties. A summary bucket is assigned a numeric weight for every node or edge mapped to it. The total number of weights for a bucket b which is also the number of resources mapped to that bucket is referred to as the bucket size $s[b]$. Concisely, $s[b]$ is defined as $s[b] = |\, d \in \mathrm{dom}(\alpha) |\ \alpha(d) = b\,|$, and S-size of buckets $(b_1, b_2, \ldots b_n) \,|\, b \in \mathrm{res}(H)$ is defined as $s[b_1] \times s[b_2] \times \ldots s[b_n]$.

Figure 4.5a shows an RDF graph for Covid-19 patients where male patients are connected to female patients, and their connections to the type of places they lived in *i.e. lyon and grenoble*. The summary S of the graph G is constructed such that X_0, and X_1 collapse into a self loop and are mapped to the same bucket b_1, X_2 and X_3 are mapped to b_3, Y_1 and Y_2, and P_1 and P_2. The size of the buckets $s[b_1] = 2$, $s[b_2] = 2$, $s[b_3] = 2$, $s[b_4] = 2$.

Table 4.8 – Summary bucket and possible expansion

(a) Summary buckets

Nodes	Size
b_1	$s[b_1] = 2$
b_2	$s[b_2] = 2$
b_3	$s[b_3] = 2$
b_4	$s[b_4] = 2$

(b) Weight

Nodes	Weight
(b_1, b_2)	$w[\langle b_1, livesIn : lyon, b_2, \rangle] = 1$
(b_1, b_3)	$w[\langle b_1, contactWith, b_3, \rangle] = 2$
(b_3, b_2)	$w[\langle b_3, livesIn : lyon, b_2, \rangle] = 1$
(b_3, b_4)	$w[\langle b_4, livesIn : grenoble, b_3, \rangle] = 2$

Let $q_1 = \{f_1, f_2\}$ where $f_1 = \langle X_0\ contactWith\ X_2 \rangle$ and $f_2 = \langle X_2\ livesIn\ P_2 \rangle$. The query can be summarized as $h_1 = \langle b_1\ contactWith\ b_3 \rangle$ and $h_2 = \langle b_3\ livesIn\ b_4 \rangle$. With the "Possible world semantics" discussed in [36], there are $\binom{s_1}{w_1}$ possible expanison of h_1 *i.e.* every expansion of h_1 is determined by the choice of w_1 triples from s_1 possibilities, the same for h_2. The product of expansion of h_1 and h_2 is the total number of expansions.

For clarity purpose, let $c = contactWith$ and $g = livesIn$. We can calculate the total expansion $\binom{s_i}{w_i}$. $\binom{s[b_1] \cdot s[b_3]}{w(s_1, c, b_3)} \cdot \binom{s[b_3] \cdot s[b_4]}{w(s_3, g, b_4)} = \binom{4}{2}^2 = 36$ possible worlds. Among the 36 possible worlds, $\binom{(s[b_1] \cdot s[b_3]) - 1}{w(s_1, c, b_3) - 1} \cdot \binom{(s[b_3] \cdot s[b_4]) - 1}{w(s_3, g, b_4) - 1} = \binom{3}{1}^2 = 9$. We say there are $\frac{9}{36} = 0.25$ possibilities. Since there are only four (4) possible expansions as shown in Table 4.8a, the expected cardinality is $4 \times 0.25 = 1$.

4.7 Result Size Estimation Problem

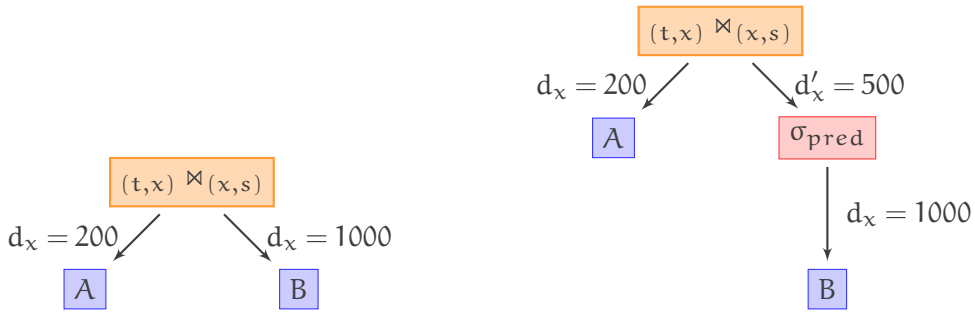
Bad cardinality estimation is an inherent problem in database systems and the effect they can have on query performance depends on the type of query and operators together with their implementation methods or evaluation algorithms. Leis et. al. [55] also reported other problems of cardinality estimation ranging from assumptions made on data distribution, methods used for selectivity estimation, join algorithms used, index usage etc.

Concisely speaking, some operators like join and filter discard tuples from the relation during query evaluation. Bad cardinality estimation is more noticable in queries with multiple joins [55]. Estimating the result size of these operators involves calculating the selectivity of the predicate (using the number of distinct values in the participating column) and multiplying that with the cardinalities of the participating relations.

Given two relations A and B, a join operation on these two relations will involve joining qualifiying columns from relations A and B. Let assume relation A contains columns (t, x) and relation A has columns (x, s) and the number of distinct value in the participating column in A is represented as d_x^A and d_x^B in B. The join size can be calculated using the formula of Section 4.2 as follows;

$$rowCount(A \bowtie B) = rowCount(A) \times rowCount(B) \times selFactor$$

Where the selectivity factor $selFactor$ is given as;



a Simple join

b Join with local predicate

Figure 4.6 – Effect of local predicate on join

$$selFactor = \frac{1}{max(d_x^A,\ d_x^B)}$$

The case described above is a simple join case as shown in Figure 4.6a where $d_x^A = 200$ and $d_x^B = 1000$. Let us assume the $rowCount(A) = 1000$ and $rowCount(B) = 1200$. According to the formula above the $selFactor = 0.001$ and join size is 1200.

Let us consider another case with a local predicate before the join operation[1]. The presence of local predicates reduces the column cardinality thereby affecting the join result size [35]. Using the same paramters (*i.e.* $rowCount(A) = 1000$ and $rowCount(B) = 1200$) as above, if we do not manage to keep track of the changes in the number of distinct values introduced by the filter, the selectivity value is estimated as $\frac{1}{max(200,\ 1000)} = 0.001$ (same as before) and the result size of the join between A and B in Figure 4.6b is thereby estimated as 1200.

However, if we consider the presence of the filter predicate in Figure 4.6b and keep track of the changes introduced by the filter before the join operation, the distinct values after the filter has been applied on B is $d_x^B = 500$. The join selectivity is estimated as $\frac{1}{max(200,\ 500)} = 0.002$ and the result size of the join $rowCount(A \bowtie B) = 2400$.

We can clearly see that if we do not consider the effect of local predicates, we will be underestimating the resulting cardinality. In order to effectively estimate intermediate results size or cardinality, we need to be able to keep track of the changes in the number of distinct values. To this end, we have adopted the work of [35] which is based on the principle of sampling without replacement, we refer interested readers to the literature [35] for more on this.

$$d' = \lceil d \times (1 - (1 - 1/d)^{|R|}) \rceil \tag{4.50}$$

where d' is the effective (expected) cardinality[2], d is the current column cardinality (without considering local predicate) and $|R| = rowCount(R)$. The interesting property of

[1] predicate that involves a single table

[2] taking into consideration the local predicate

this formula is that when $d = |R|$, $d \times selFactor$, when $d \ll |R|$ the value returned is closer to d.

4.8 Propagating Cost Parameters Through the Query Tree

We have discussed the cost formulas for each relational operator of [26] in Section 4.4. Estimating the cost for these relation operators is straight-forward, however, a query plan is a sequence of relational operators (with data dependency) is chained to form a tree; a *QEP*. The idea of a cost model is to be able to propagate statistics and cost parameters so that a changes in size, cost and any of the statistical parameters are capture and utilized by the cost model to make accurate cost estimation.

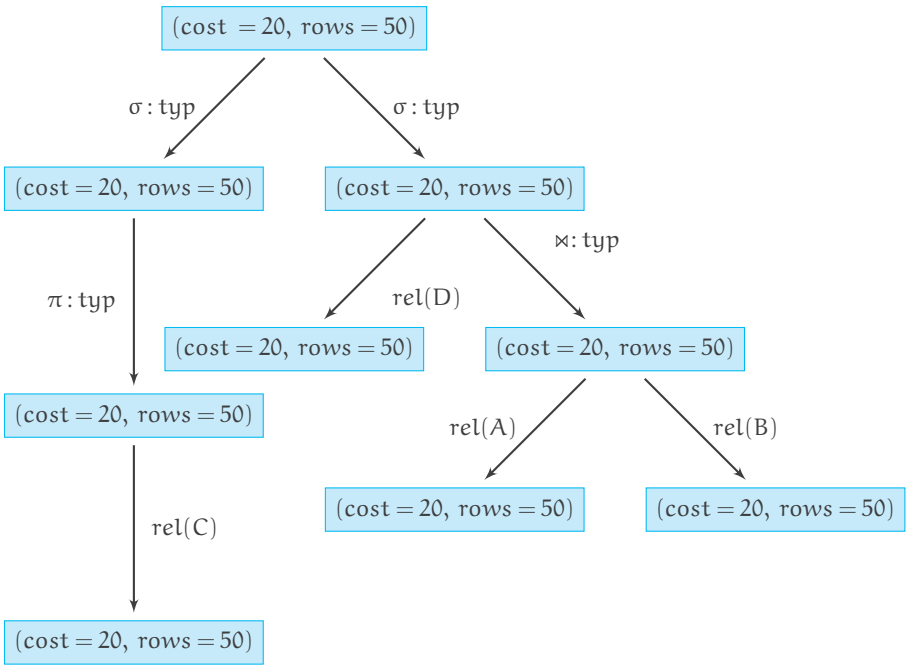


Figure 4.7 – Cost Estimation Graph (CEG)

The cost and statistical properties on the branches are propagated through the Cost Estimation Graph (*CEG*). Figure 4.7 shows a *CEG*, a tree representation of the cost information or properties of operators in a given *QEP*. In a *CEG*, links between nodes shows the dependencies and connections between parent and child operators.

Cost and data statistics propagation started from bottom of the *CEG* to the top. The cost of a particular branch of the CEG is the summation of the costs of each child node belonging to that branch. The cost of each branch is summed as the total cost of the *QEP*.

$$Cost(QEP) = \sum_{P \in \varphi} (evalCost(P), rowCount(P)) \tag{4.51}$$

Given a *CEG*, we estimate the cost of a query plan by adding together the evaluation cost ($evalCost$) of all the child operators of the query tree. The cost of a query plan is

given in Equation 4.51. Similarly, the total result size ($rowCount$) is obtained through statistics and cost propagation and the total correspond to the addition of all the operators result sizes taking into consideration the selectivities and reduction factors.

4.9 Summary

In this chapter, we discussed cost estimation techniques for recursive queries. We started the chapter from the basic knowledge of cost estimation techniques available in the literatures for non-recursive physical query plans, adapting it first to relational algebraic operator (LQEP) and deriving a novel cost estimation technique for fixpoint operators.

The components that made up the cost formulas like selectivity and cardinality were discussed comprehensively and we showed how they are combined with other parameters to form the cost formula. In Table 4.9, we present the summary of the cardinality and cost formulas for the operators considered in this work.

Table 4.9 – Cardinality and cost formula for μ-RA operators

#	**Operator**	**Cardinality** ($rowCount$)	**Cost Formula** ($evalCost$)
Let $c = cost(\varphi)$, *cost* is given in Section 4.3.1.			
1.	E	$rowCount(E)$	$relPages(E) + rowCount(E) * w$
2.	X	1	1
3.	$\lvert c \to v \rvert$	1	1
4.	$\rho_a^b(\varphi)$	$c.rowCount \times rFactor$	$c.rowCount$
5.	$\widetilde{\pi}_a(\varphi)$	$c.rowCount \times rFactor$	$c.rowCount$
6.	$\sigma_f(\varphi)$	$c.rowCount \times selFactor$	$c.evalCost + c.rowCount$
Let $c_1 = cost(\varphi_1)$, $c_2 = cost(\varphi_2)$ and $selFactor$ is the selectivity factor			
7.	$\varphi_1 \cup \varphi_2$	$c_1.rowCount + c_2.rowCount$	$c_1.evalCost + c_2.evalCost$
8.	$\varphi_1 \bowtie \varphi_2$	$c_1.rowCount \times c_2.rowCount \times selFactor$	$c_1.evalCost + (c_1.rowCount \times c_2.evalCost) +$ join size
9.	$\varphi_1 \triangleright \varphi_2$	$c_1.rowCount \times c_2.rowCount \times selFactor$	$c_1.evalCost + (c_1.rowCount \times c_2.evalCost) + c_1.rowCount$
Let $c_1 = cost(\varphi)$, $c_2 = cost(R \bowtie X)$, $\mathbb{N}$ = number of iterative steps			
10.	$\mu X \,.\, \varphi \cup (R \bowtie X)$	$c_1.rowCount + rowCount(R \bowtie E)^{\mathbb{N}}$	$evalCost = c_1.evalCost + (c_2.evalCost \times \mathbb{N}) + rowCount$

Chapter 5

Effectiveness of Cost Model QEP Selection

Like with many other systems, validation of cost model in database query optimizer is crucial to identifying and improving the accuracy and quality of plan selected by the query optimizer.

5.1 Introduction

Performance of query engines is an important factor for choosing appropriate database system for query evaluation [87, 101]. Query engines performance are mostly measured in terms of it plan-picking function through it cost model. As presented in Chapter 3, there are a variety of cost models depending on the query engine and their optimization approach and the type of operators they support.

A typical query optimizer consists of different components that work together to generate efficient ways for evaluating a given query. Validating query optimizers quality independently remains an open challenge as researchers have focused on one aspect of the optimizer than the other. Instead of assessing the effectiveness of the optimizer's term (or plan) selection and the quality of the selected query plans through its cost estimation, some existing benchmarks like TPC benchmarks assesses the query runtime system, JOB [55] benchmarks research in cardinality estimation and join order optimization, [93] provides a robustness metric for plan selection with respect to the cardinality estimation error.

In order to address these challenges, we present a framework for assessing query optimizers cost models quality. By decoupling query optimizer components, we isolate the cost model from other components of the query optimizer and focus our attention on how they can be validated regardless of the underlying system or architecture. In the next section we define the problem statement and we go on to explain the optimality

conditions for query plans, thus guiding the validation of cost models using these set of conditions.

5.2 Problem Statement

As previously explained in Chapter 4, cost estimation relies on statistics of input data and query to make accurate estimation. In practise, many query optimizers choose query plans that are inefficient and a plethora of query optimization algorithms and cost estimation techniques exist.

The least cost query plan according to the cost model does not always imply the least actual runtime plan. We are at risk of missing out on quality query plans. For example, if we are given a query with 100 equivalent plans and two cost models A and B. Let us assume that A selects a plan whose actual observed runtime is the 20^{th} cheapest and B selects a plan whose actual runtime is the 21^{th} cheapest, based on current cost comparison methods, A is a better cost model of the two, this is true. But at the same time what happens to the 19 other cheapest plan and how exactly does these cost rank in terms of their plan-picking capabilities?

Many query engines' cost estimation techniques exhibit changes in plan-selection behaviour when certain input paramters are slightly adjusted. Wu et al. [32] showed that calibrating cost paramters in PostgreSQL database results in improvement in query runtime. In contrast to that, Leis et al. [55] demonstrated that using the true cardinality of query often improve the cost estimation and quality of the plan selected by query optimizer, suggesting that even the simplest cost formulas in the presence of accurate cardinality will be sufficient enough for any query optimizer.

Additionally, query optimization and execution are a complex combination of several factors like storage models, optimization algorithms, data or result caching etc. Query engines are different and they have different optimization techniques implemented, hence, execution of queries on different query engine differs, even for sub-optimal query plans some query engine performs relatively better than others. Some have better cost model but not the fastest execution or optimization algorithm implemented i.e. they have moderate speed of execution for query plans. As a result, direct comparisons of query execution time from different optimizers might not accurately give a clearer understanding of the effectiveness of the cost model.

For learning-assisted cost estimation or learning-based plan selection techniques, data about previously run query, plan cost and workload are important data that this method use to learn and further improve estimation accuracy, meaning that for changing workload and query, there might be need to train and re-train. Until now, there has not been a structured data model and repository for collecting information like this.

In general, we note a lack of standard method for validating cost models in query optimizers. To this end we present a cost model validation framework in this chapter.

Definition 5.2.1. ***(Actual Cost).*** *of a query plan refers to the total time used by a query engine to evaluate a given query plan. We interchangeably use "True" and "Actual" cost throughout this work. Actual cost of a query plan* P *is denoted by* $\texttt{cost}(\texttt{P})_{\texttt{true}}$

Definition 5.2.2. ***(Exact Cardinality).*** *refers to the actual number of tuples or nodes returned by an expression, an operator or a query plan obtained by an actual execution of a given query plan.*

5.3 Metrics and Specifications

In this section, we discuss the metrics that measure the efficiency of a cost model and we also present a set of standard specification for a cost model.

5.3.1 Metrics

What often comes to mind is what do we measure the validity of a cost model by. In this section, we define a set of metrics that will help us validate cost models by answering several questions.

- Are query plans selected by the cost model always optimal and how do we determine optimality of query plan?
- What is the impact of estimation errors on query plan quality?
- Are there any correlation between cost variables and query execution time?

5.3.2 Specifications

A framework for assessing a query optimizer cost model must satisfy the following requirements.

- **Quality of Plans Selected:** the cost validation framework must be able to assess the effectiveness of a cost model with respect to the quality of query plans they choose. This means that the cost validation framework must use the quality of the plan selected as one of the many paramters to assess the effectiveness of the cost model.
- **Isolated Assessment:** cost validation framework should be able to assess the cost model in a decoupled manner from the query optimizer plan space generation for instance. This will facilitate accurate comparisons between different cost models and this will give users the freedom to easily extend the framework.
- **Extensibility:** the framework should be designed in such a way that new features or functionalities can easily be added.

- **Uniformity and Generality of Configuration:** an assessment framework should be able to compare query optimizer cost models using uniform configuration settings in order to ensure that every system tested at the same time are given equal chances to prove their effectiveness. For example, if a given timeout is given to a system A, the same amount of timeout should be used to test other systems it is being compared to. For query engine running on different platform, metrics that are incomparable on all platforms needs to be avoided. Irrespective of the platform, the framework should be easy to setup and run with minimal effort.

5.4 Optimality of Query Plan

The cost model returns a list of plans with their respective costs as the CEG. The plan with the minimum cost is termed the "optimal plan" and this plan is sent for execution.

Definition 5.4.1. *(**Optimal Plan**) Let* P *be a plan space such that* $P_1, P_2, \ldots P_n \in P$ *be the equivalent query plans for a query* Q, $evalCost(P)$ *represents the application of cost function on the plans in the plan space. We say a plan* P' *is the optimal term iff;*

$$evalCost(P') = \arg\min(evalCost(P)) \tag{5.1}$$

Definition 5.4.2. ***(Near-optimal plan)**. Given a plan space for a query* Q *containing a set of equivalent plans* $P_1, P_2, \ldots P_n \in P$*, with actual costs (i.e the total time taken to run the query and return results), there exist a plan* P_i *with actual query time or cost that is larger than the optimal plan cost but whose cost still falls within the first quartile range of the costs of all the plans considered in the plan space.*

Let $T(P)_{nopt}$ be the actual cost of the near optimal plan, $T(P)_{min}$ is the cost of the optimal plan and $T(P)_{25th}$ be the cost in the first quertile,we say that;

$$T(P)_{min} \leqslant T(P)_{nopt} \leqslant T(P)_{25th} \tag{5.2}$$

With near-optimal plan we can be sure that the selected query plans are always good enough and as competitive in terms of actual query time. Contrary to the definition given in [93] where a near optimal plan is deemed to be at least 1.2x times larger than the estimated optimal plan, we believe that this value might become too big in many cases making query plans to be more sensitive to estimation errors and as such, the better performance cannot always be assessed.

Figure 5.1 illustrates the cost of all equivalent plans for a given query. The cost is ranked by obtaining the percentile (further discussion on the percentile is in later section of this chapter) of all the costs. We start by getting the plan with the minimum actual cost

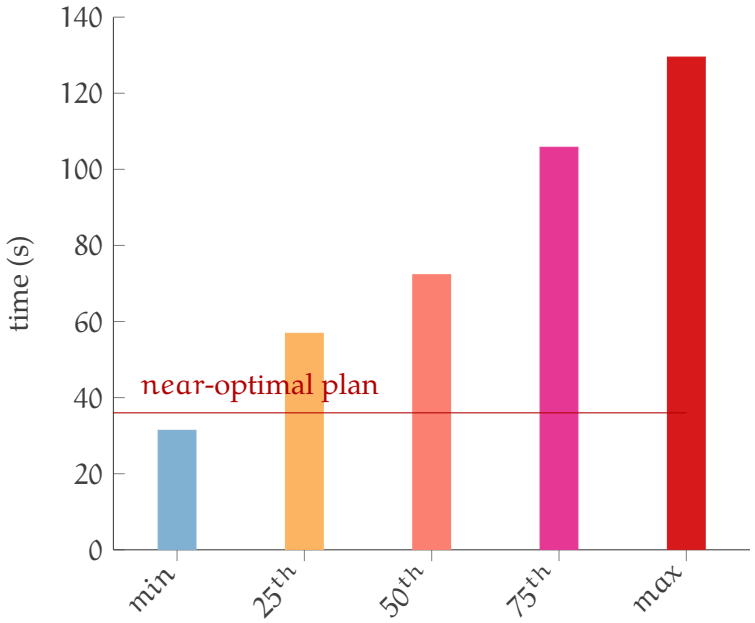


Figure 5.1 – Cost ranking showing optimality of plan

and getting the quartiles and maximum values (max) for the costs. A *near-optimal* plan should have its cost between the minimum (min) and 25^{th} percentile at all times. The horizontal line on graph in Figure 5.1 represents the *near-optimal* plan cost. Generally, the closer this value is to *min*, the better it is.

5.4.1 Condition for Optimality

For any given set of query plans of a user-defined query, we define the following conditions that needs to be satisfied in order to guarantee optimality.

1. **100-Plan**: all (100%) of the plans generated from the search space (under the given timeout period) are considered. This condition ensures that in most cases, good plans are not left out during the search for optimality (or near-optimality) in case the optimizer poorly estimates the cost of the plans. If the optimizer's cost model is fairly accurate, this rule can be relaxed to 90% or lesser in order to reduce evaluation time.

2. **Least-cost Plan (LCP)**: a query plan with the least cost is always considered as the cheapest. In the presence of estimation errors which is mostly inevitable in query plan cost estimation, we expect that the actual cost of the plan selected always falls within the minimum actual cost and the 25^{th} percentile of the total cost of all the plans in the query plan space. The closer the value to the minimum, the better.

3. **Least Cost Error (LCE)**: an optimal query plan must be a plan with the minimum cost estimation error among all the query plans cost estimation errors, with or without using an actual cardinality during cost estimation. Similarly, a near-optimal

plan must be a plan with cost estimation error closer to the minimum cost estimation error. We discuss cost estimation errors later in this chapter.

4. **Cost Variables Correlation (CVC)**: there is correlation between the estimated cost and the actual cost or runtime of the plans generated for any given query. What this means is that, the higher the estimated cost of a query plan the higher the actual time taken by the query engine to evaluate the plan.

5. **Logical Operator Ordering (LOO)**: to push things a bit further, this condition ensures that logical precedence of operator is observed in the choosen query plan. As an example, a filter (σ) operator is expected to be pushed close to the source node as possible in a query tree. This condition also serves as some kind of stress test for the cost model since it allows for the query plan selected by the cost estimation function as the cheapest to be compared with some pre-defined set of rules or know pattern that guarantees optimality.

We discuss more on conditions *2, 3, 4 and 5* below. We have left out the first condition 1 (100-plan) as we believe this conditions is clear and straightforward.

5.5 QEP Cost Estimation Ranking

Another important way to validate a cost model is by ranking the term or plan picking function of the cost model. Given a query Q with a set of equivalent query plans $P_1, P_2, \ldots P_i \in P$, a term or plan picking function selects the best plan based on the cost estimation from a pool of plans generated in the plan space.

In order to assess how a cost model qep-picking function compares to the best query plan of P (i.e the ones with the minimum observed query runtime) and compute the cost estimation ranking, we normally need to run all equivalent query plans from the plan space P that are generated by the optimizer.

Ranking cost models with respect to query plans gives an insight into the closeness of the cost model to reality i.e. the closeness of the cheapest plan selected by the cost model to the actual optimal plan based on measured runtimes. Our ranking approach is based on interval of estimates using percentile ranking to determine in which percentile the plan chosen by the cost model lies. There are seven ranking categories in total; min, 15th, 25th, 50th, 75th, 90th and max. When comparing two cost models, the percentile analysis shows how good or bad the cost estimation function is. A better cost model chooses more query plans that rank in the lower percentile range.

Recall, in Equation 5.2, we define the condition for a near-optimality where we set the upperbound for the ranking to be in the 25th range. We note that this range is fairly large and a lower range like 15th percentile will be better if there is high variation in the query time of the plan spcae.

If P represents the set of plans generated in the plan space, T is the runtime for a given plan, we define a ranking threshold function Φ as follows;

$$\Phi = \begin{cases} 1 - \frac{T_{min}(P)}{T_{25th}(P)}, & \text{if } T_{min} < T_{25th} \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

where $T_{min}(P)$ is the plan with the minimum actual runtime of all the plans in the plan space and $T_{25th}(P)$ is the runtime in the 25th percentile range. Φ is between 0 and 1.

Definition 5.5.1. ***(Cost Model Rank).*** *A cost model rank (for a plan) is the factor by which the execution time of a plan P^* selected by a cost model is greater than the plan with the minimum query execution time in the same plan space and less than the ranking threshold for that plan space.*

Let λ_P be the factor by which the query evaluation time of a plan P is greater than the minimum query evaluation time of a given plan space.

$$\lambda_P = 1 - \frac{T_{min}(P)}{T(P^*)} \tag{5.4}$$

The cost model rank for a set of queries Q_n is the number of times the ranking factor of the best plans choosen by the cost function is equal to or less than the ranking threshold for their respective queries. For a set of queries Q_n, a cost model cm is ranked wrt Φ;

$$Rank(cm) = |\lambda_{P \in Q_n}| \qquad \text{such that} \quad \lambda_{P \in Q_n} \leqslant \Phi_{P \in Q_n} \tag{5.5}$$

Table 5.1 consists of cost model ranking from [20] for two different cost models; System P and System P′ over 20 queries. System P selects 6 query plans for six different queries whose actual cost is the minimum of all the plans generated from the plan space for each query. This means that the cost model selects optimal plans in 30% of total number of queries compared to 35% (7 total minimum pick) of System P′.

Table 5.1 – Cost model ranks with number of plans in each rank

Cost Model	min	15th	25th	50th	75th	90th	max
System P	6	-	-	1	11	1	1
System P′	7	4	4	2	1	1	1

If we consider a near-optimal range of 25th percentile, we can see that System P′ is 55% of the time pick near-optimal plans compared with 30% for System P. We can clearly say that System P′ is a better cost model than System P based on the ranking presented in Table 5.1.

5.6 Estimation Errors

Estimation errors are important in predicting the correctness of cardinality and cost estimation of query plan. Given a set of query plans P, belonging to a query Q, an efficient cost model will aim to select a plan P_i that minimizes the estimation errors. In this section, we divide the estimation errors into two (2) categories; cardinality and runtime cost estimation errors.

5.6.1 Cardinality Estimation Errors

For any given query Q, we consider the q-error which measures in absolute the deviation of estimated cardinality ($rowCount_{est}(Q)$) from true cardinality ($rowCount_{true}(Q)$). The cardinality error is tested per cost model for each queries.

$$\text{q-error}(Q) = max\left(\frac{rowCount_{true}(Q)}{rowCount_{est}(Q)}, \frac{rowCount_{est}(Q)}{rowCount_{true}(Q)}\right) \tag{5.6}$$

In Equation 6.3, we define the q-error to be the upperbound of the ratio of the true and estimated cardinality following the work of [102]. Usually, the closer to 1 the estimation error is, the better it is.

We divided the q-error into intervals where the first interval is always [0, 1] and it means that the estimation is accurate i.e the number of tuples returned by the cardinality estimation framework is the same as the true cardinality.

Table 5.2 – q-error interval

q-error interval	Frequency		
	Dataset 1	Dataset 2	Dataset 3
[0, 1]			
[1, 2]			
[2, 4]			
...	...	...	...
[n-1, n]			

Table 5.2 shows an example q-error interval. When comparing optimizers' cardinality estimation accuracy, the model that has more items in the lower interval range has a better cardinality framework. The table can be represented as a graph.

The formula in Equation 6.3 measure the error property of the cost model cardinality estimation but there is no way to tell if this is the result of overestimation or underestimation. To this end, we redefine q-error to capture this properties and we present estimation error $s'(Q)$ for a query Q in Equation 5.7.

$$s'(Q) = \begin{cases} |\frac{\hat{c}-c}{\hat{c}}|, & \text{if } c > \hat{c} \\ \frac{c-\hat{c}}{c} & \text{otherwise} \end{cases} \tag{5.7}$$

where $c = rowCount_{true}(Q)$ and $\hat{c} = rowCount_{est}(Q)$

We have two cases of cardinality estimation errors according to Equation 5.7;

- Under-estimation: (case 1 in Equation 5.7) when the cost model under-estimates the cardinality of a query plan, the q-error is a positive value greater than 0.
- Over-estimation: (case 2 in Equation 5.7) a cost model can over-estimate the cardinality and when it does the q-error is less than 0 i.e. a negative value

The plan that minimizes the q-error for the cardinality estimation is expected to be the best plan or at least a near-optimal plan because cost estimation significantly rely on the accuracy of the cardinality estimates [55] but as we will show in the next chapter, this assumption does not always hold in practise if the cost estimation is not well-planned and formulated.

5.6.2 Cost Estimation Error

As previously mentioned, the cost returned by the cost model for any given plan is only an a priori estimation and not the exact measure of the true observed cost. We introduce two error metrics (i) c-error and (ii) mean realtive error (MRE) to measure the deviation of the estimated cost of query plans from the actual cost (the real measured ones).

5.6.2.1 c-error

This cost error is used to compare the closeness of the estimated cost for a plan compared to the actual query execution time for that plan. This metric enables us to refer to the return value of a cost model as either absolute or relative measurement. An absolute measurement means that the estimated cost is close in value to the execution time for most cases and they are given as time value. Relative measurement refers to an estimated cost having a large disparity with the actual execution time of a plan.

We apply a variation of the q-error which has been previously described in [55] and [36] used mostly for cardinality estimation error as the cost estimation error.

If we denote $evalCost(P)$ as the estimated evaluation cost of a term φ consisting of a sequence of RA expressions, let $exactCost(P)$ denote the exact cost (measured as the query time). We define the cost-error for the evaluation cost as;

$$\text{c-error}(Q) = \| \frac{exactCost(P)}{evalCost(P)} \|_Q$$

Let $P = \{p_1, p_2, \ldots p_n\}$ denote a set of equivalent plans generated from the plan space for a term φ. Let p^* be the best plan in P minimizing $exactCost(p)$ and p' be the cheapest plan selected in P according to the cost estimation for a query Q. We now focus our attention on the different factor between the exact cost and the estimated cost.

Theorem 5.6.1. *If for all* $p_i \in P$

$$\| \frac{evalCost(p_i)}{exactCost(p_i)} \|_Q \leqslant q \tag{5.8}$$

for some q, then

$$\| \frac{exactCost(p_i)}{exactCost(p_i)} \|_Q \leqslant q^2 \tag{5.9}$$

In Theorem 5.6.1, we give an upper bound of the factor difference between the exact and estimated cost and the precision of our cost model is given up to q factor. The query plan selected by the cost model is erroneous up to q^2 away from the actual optimal plan (derived by running all query plans).

Proof. We say that p' is the cheapest plan under the evaluation cost function $evalCost$, then we must have

$$evalCost(p') \leqslant evalCost(P^*)$$

recall that p^* is minimal under $exactCost$, then

$$exactCost(p^*) \leqslant exactCost(p')$$

Since for any plan p, we have $\| \frac{exactCost(p)}{evalCost(p)} \|_Q \leqslant q$

$$exactCost(p') \leqslant q \cdot evalCost(p')$$

$$exactCost(p^*) \geqslant \frac{1}{q} evalCost(p^*)$$

We then derive the following

$$\begin{aligned} \| \frac{exactCost(p')}{exactCost(p^*)} \|_Q &\leqslant \frac{exactCost(p')}{exactCost(p^*)} \\ &\leqslant \frac{q \cdot evalCost(p')}{(1/q)\, evalCost(p^*)} \\ &\leqslant \frac{q \cdot evalCost(p^*)}{(1/q)\, evalCost(p^*)} \\ &\leqslant q^2 \end{aligned}$$

■

5.6.2.2 Mean Relative Error

Another metric we considered is the Mean Relative Error (MRE). MRE is a metric defined in [58] and it is used to minimize the relative cost estimation error in all queries regardless of their execution time. This metric is useful for comparing two cost models. Comparison

is achieved by taking the relative error (RE) for each query and using this to calculate the MRE.

$$RE = \frac{evalCost(Q_i) - T_i}{T_i} \tag{5.10}$$

In Equation 5.11, $evalCost(Q_i)$ is the estimated cost of a selected plan for a given query Q_i while T_i is the actual execution time for the query. MRE is always a positive number.

$$MRE = \frac{1}{N}\sum_{i=1}^{N}\frac{evalCost(Q_i) - T_i}{T_i} = \frac{1}{N}\sum_{i=1}^{N} RE_i \tag{5.11}$$

where N is the number of queries.

Again, the MRE is used to validate the effectiveness of the cost model. Lets assume we have two cost models A_1 and A_2 having MRE_1 and MRE_2 respectively. If $MRE_1 < MRE_2$, then we say A_1 minimizes the MRE thus being a better cost model of the two.

5.7 Cost Metrics and Query Time Correlation

Cost variables correlation test the relationships between various cost metrics. Correalation is a statistical method and there are three different correlation types as shown in Figure 5.2; (i) positive correlation, in which case the values on the y-axis increases as the values on x-axis increases (ii) negative correlation, where the values on the y-axis decreases as the values on x-axis increases and (iii) no correlation, where the the relationship is not identifiable and relationship can not be fully established between the two features.

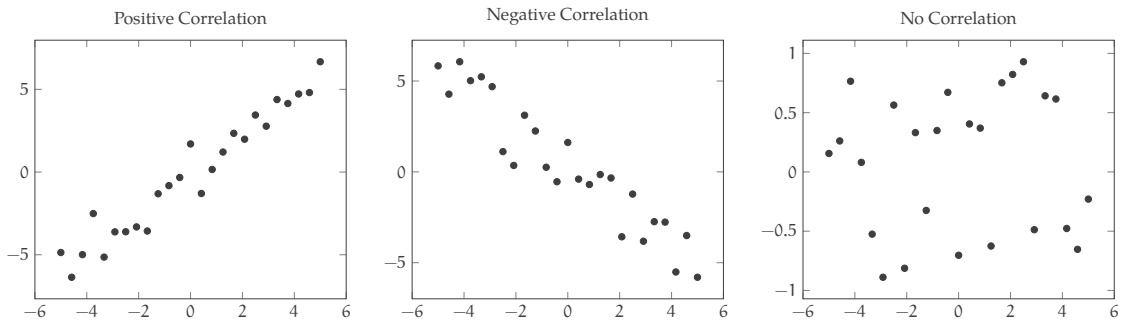


Figure 5.2 – Correlation

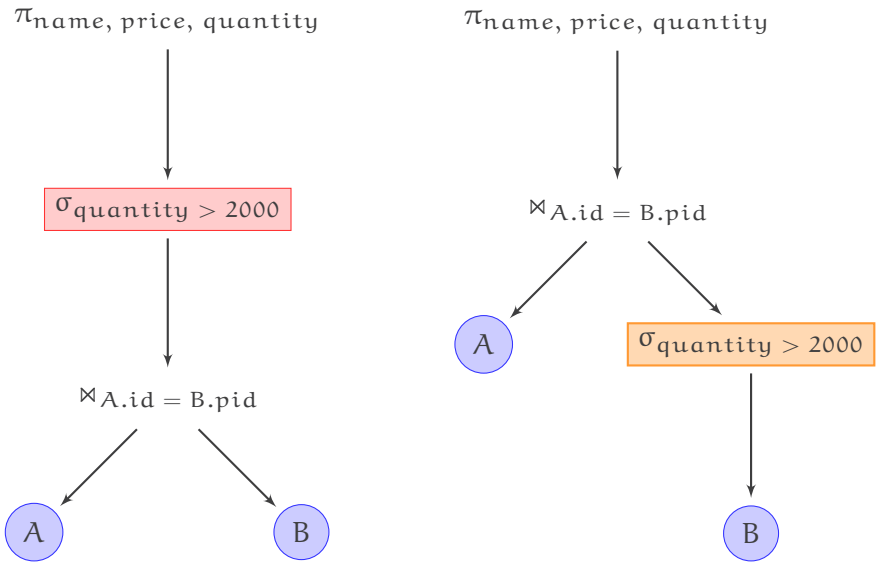
Depending on the metrics between which we try to establish the correlation type might have different interpretations. Some of the cost metric that will be interesting to establish a correlation between include;

- q-error vs. execution time
- cost ranking vs. execution time vs. plan quality
- MRE vs. average execution time
- cost error vs. execution time

A positive correlation between the q-error and query execution time implies that estimation error affects query execution time in such a way that the higher the estimation error, the slower the query execution is. A strong positive correlation might also tell us if the cost model underestimate the cardinality. A strong negative correlation point towards a cardinality overestimation by the cost model. In general, we expect a strong positive correlation between the cost rank λ_{CM} and the plan quality, indicating that the higher the cost ranking threshold is, the slower the query execution time is and the less quality the query plan is selected by the given cost model.

5.8 Operator Ordering on the Query Tree

As mentioned in the previous chapter, a query tree is an ordered sequence of operators or sub-expressions chained together to perform a single task of query evaluation and return results (if any) upon completion. How these operators are combined in a query tree has a significant influence on the query runtime and resources utilization.



a QEP A: plan selected by cost model

b QEP B: Optimal plan

Figure 5.3 – Equivalent plans with different ordering of operator

We show a simple example of operator ordering in Figure 5.3. The query retrieves product names, prices and quantities whose quantity of products is greater than 2000. The most natural way to write this kind of queries is QEP B where unnecessary rows are removed prior to the join operation, reducing the number of rows unlike QEP A where the two relations A and B are joined before removing any product with a quantity less than 2000. It should be noted that this operator ordering we see here is a common logical optimization but it is helpful to check this ordering in case of cost estimation errors where a different plan (say QEP A) might be deemed to be the cheapest.

By comparing the query tree of the plan selected by the cost model in Figure 5.3a with an expected ordering predefined as the optimal plan in Figure 5.3b, we could easily tell that the selected plan is not optimal. Also, this approach could be useful for supervised learning-based method where this type of ordering can be encoded into the training model.

5.9 Modeling Cost Validation Framework

So far, we have discussed the important conditions and metrics for testing to the limit, the optimality of query plans and the validation of an optimizers' cost model. In this section, we propose a cost validation framework for storing information about queries together with their respective execution plans, cost models and the results of their validation.

A data model for the cost validation framework is presented in Figure 5.4. This model describes the data types and relationships that exist between the different entities. The model provides a convenient way to assess the plan-picking capabilities of a cost model and also allows comparing different cost models.

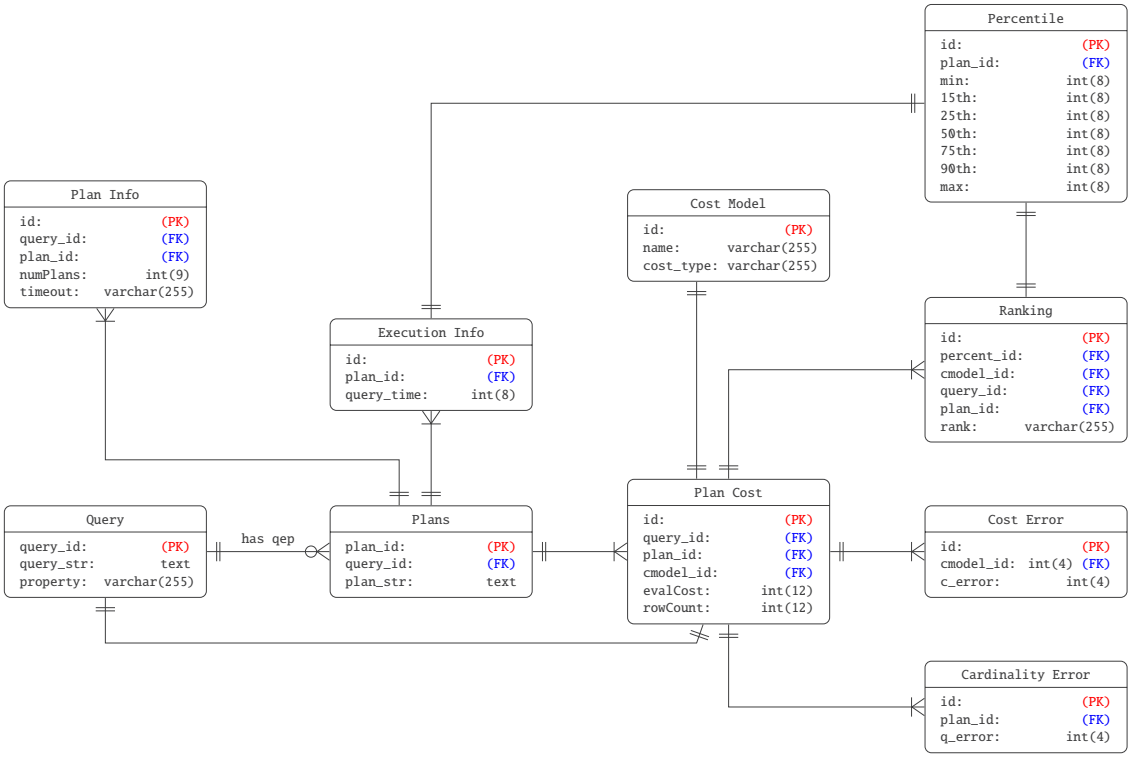


Figure 5.4 – Cost validation model

While Ouared et al. [99] proposed a domain-specific language for designing cost models which designers can adapt to collaboratively callibrate cost model paramaters, our work here is very different in that it is focused on building a metadata repository for cost validation and plan quality improvement. This helps to give structure to an already challenging cost model validation. In addition, where learning-based method is deemed

appropriate, this framework can serve as a data store where information about query execution can be stored for future workload to learn from. We do not provide a specific language for the framework but it is our believe that a structured model like the one presented here has provided a foundation that makes it easier to develop a language to interact with this framework.

5.10 Summary

We started the chapter by describing the metrics and specifications for our cost model validation framework. In Section 5.4, we define what is an optimal plan and specify the set of conditions that guarantees the optimality of query plan. Each of this conditions are described in further details in a later section. Metrics such as cost ranking, estimation error, cost metric correlation and validating operators depth on query tree are important factors that help in measuring the consistency and impact of such metrics on the plan-picking and the effectiveness of a cost model.

We then present the data model for the cost validation framework, although, we do not provide a language for the framework, we believe that the specifications is helpful for the design of an evaluation method for query optimizers.

Chapter 6

Implementation and Evaluation

6.1 Introduction

Nowadays, one of the biggest challanges is estimating the cost of recursive query plans. Even for the simplest case of recursive queries, most data processing systems rely on several assumptions and magic constants in order estimate the cost of recursive operators or in some cases no estimation is done at all and the query planner just rely on heuristics for the optimization of this class of queries. In the previous chapter we present a cost estimation technique for recursive relational algebraic terms for MuRA [26]. Query and plan space generation is beyond the scope of this work and as a result of that, we rely on existing infrastructure of MuRA.

To this end, we will now focus on the implementation of the cost model discussed earlier for a broader set of regular path queries (RPQs) which covers the conjunction (CRPQ) and union (UCRPQ) of this class of queries in their translated RA forms. We focus on logical query plans (LQEPs) generated from path queries and discuss the different components of our cost model.

In Section 6.2, we present the cost model implementation. We start by giving an architectural overview of the cost model components then proceed to discussing each component. The statistics, their collection method and design choices are also discussed in the section. In Section 6.3, we discuss the integration of state-of-the-art cardinality estimation for RDF graphs into our cost model in order to improve the accuracy of the estimation. Estimating the cardinality of subexpressions on a query plan needs to be carefully carried out in order to avoid estimation errors during query decomposition and translation from one form to another, we discussed this in Section 6.3.

In Section 6.4, we present all the techniques used to assess effectiveness of our cost model, this include evaluation of our cost function on different dataset and queries, the

effects of the cardinality estimation technique used, percentile analysis of the cost model showing how we ranked against the state-of-the-art.

6.2 Implementation

Since query rewriting is beyond the scope of this work, we restrict our focus on the cost estimation aspect and we give the implementation details of the different components of the cost estimation framework.

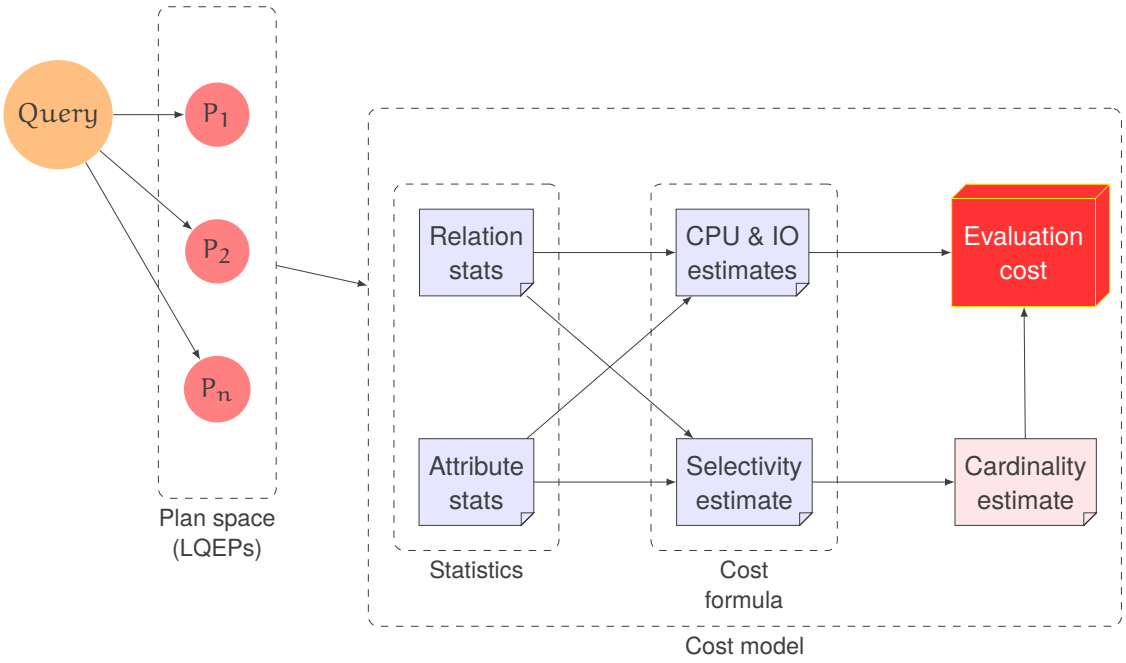


Figure 6.1 – Cost model architecture

The architecture of the cost model used in this work is shown in Figure 6.1. Relation and attribute statistics together with query plans are fed as input parameters to the cost formulas which are data that are either retrieved if they are base relations or calculated if they are intermediate operators. The cost function calculate two factors; (i) the CPU and IO costs and (ii) the selectivity to track the changes in cardinality. The final cost returned by the cost function for a query plan is an estimate of the evaluation cost and the cardinality.

6.2.1 Obtaining Base Relation and Attributes Statistics

As stated in Section 4.2, the cost model makes use of statistics in order to estimate the cost of a given query plan. These sets of statistics are broadly divided into two; the relation and column statistics. The relation statistics which include the number of tuples in the relation, number of pages that hold the tuple of the relation. These statistics are available in the pg_stats class of PostgreSQL (with a first run of the ANALYZE command) or computed by counting the number of tuples for systems where these information is not available. The

number of pages that holds the tuple of the relation can be calculated using the number of tuples, the page size and the maximum size of strings.

The other set of statistics is the column statistics which include the number of distinct values per attribute of the relation, the histogram of frequencies of the attributes in the relation. Distinct column count and histograms are useful for result size estimation. Histogram partition values into buckets which stores frequencies (mcf) of attributes values present in the corresponding intervals, most common values (mcv) in the interval (number of values in the intervals), number of distinct values and the number of NULL-values. We rely on the equi-width histogram similar to the one used in PostgreSQL.

During cost estimation, the cost model looks through the histogram to retrieve the selectivity of a given predicate. Remember there are only two cases of filter predicates considered in MuRA and discussed in the previous chapter. When the selectivity value is not present in the histogram bucket, we apply the formula in Equation 6.1 which assumes a uniform distribution of data.

$$selFactor = (1 - Sum(mcf)) \ / \ (D_i - Count(mcv)) \tag{6.1}$$

where $Sum(mcf)$ is the sum of the most frequent values, D_i is the number of distinct values for the column and $Count(mcv)$ is the total number of mcvs.

Whenever the histogram data is not available the cost model falls back to the formula described in Section 4.2 from [7].

6.2.2 Cost Formulas and Query Plan Costing

MuRA translates a path query (U/C/RPQ) queries to its internal representation; the relational algebra (RA) and undergo different rewriting techniques in a bottom-up manner. In the plan space, different equivalent plans $P_1, P_2, \ldots P_n$ are generated for each given query Q.

The query optimizer returns a list of equivalent logical query execution plans (LQEP), each of which is a tree corresponding to the operation to be performed. The list of LQEP is sent to the cost model where each operator is given a cost using the mathematical formulas presented in Chapter 4 and the cost for each operator is propagated and added to form a final cost for each LQEP. Cost estimation for query plans are done in parallel in order to speed up estimation. The cost model returns a list of plans with their respective costs as the CEG. The plan with the minimum cost is chosen as the "best plan" and is forwarded for execution.

Cost estimation is done by combining the CPU and IO cost factors. The cost of a given query plan is recursively computed such that the children expressions or operators and the root are summed to arrive at a final cost for the plan. At each step, every operator estimate the evaluation cost $evalCost$ and the cardinality $rowCount$. Operators could be unary (e.g. rename, filter, antiprojection), binary (e.g. union, join) or a fixpoint operator and their cost are computed using the cost formula described in the previous chapter but

the cost vary from one plan to another depending the query, statistics and the arrangement of operators or expressions in the query tree.

The cost estimation given here is modeled in terms of the relative time taken to execute a given query plan.

6.2.3 Design Choices

Some design decisions were taken for the implementation of the cost estimation techniques of this work. They include the choice of fixpoint evaluation algorithm and the query optimizer we have chosen.

The cost formula for the fixpoint operator described in Chapter 4 is based on a semi-naive recursive query implementation. The semi-naive evaluation of fixpoint is more efficient than the naive method in that ensures that redundant re-computation of datasets is avoided by generating new sets of tuples from the result of the last iteration result. Apart from being applicable to a simple case of linear recursion where reference to recursion is only made once to itself, the techniques introduced in MuRA [26] facilitates having nested fixpoint operators and the technique introduced here are adapted for both kinds of transformation rules.

At the moment, we employ an explore-first-cost-later approach in which the optimizer first generates a list of possible equivalent plans and these plans are sent to the cost model afterwards for cost estimation as opposed to using the cost to prune the plan space which is characteristic of Volcano-style [50, 51] optimizers. To this end, we plugged our technique into the bottom-up System-R [7] style optimizer. The techniques proposed in this work in general can be applied to any query optimizer.

6.3 Integrating SumRDF Cardinality Framework

SumRDF [36] is a cardinality estimation technique for non-recursive SPARQL queries over RDF graphs. To facilitate the integration of this technique for recursive query cardinality estimation over graphs there is a need to ensure seamless conversion and decomposition of queries. The cost estimation techniques presented here is primarily targeted for the logical phase of query optimization. We decompose and translate complex RA expressions into a dialect supported by SumRDF.

As stated earlier, we are dealing with complex recursive queries that contains at least one union, multiple joins and fixpoint operators. Basically, we unnest regular path queries to have a finer grain of the cardinality estimation and produce a set of queries that conform to the SPARQL v1.0 standard used in SumRDF.

In Figure 6.2 we show how we design our cardinality estimation framework by integrating SumRDF and annotating query plans with information with the cardinality of sub-terms or expressions. When a recursive query is given, we first decompose the recursive query into non-recursive SPARQL queries and estimate the cardinality with SumRDF.

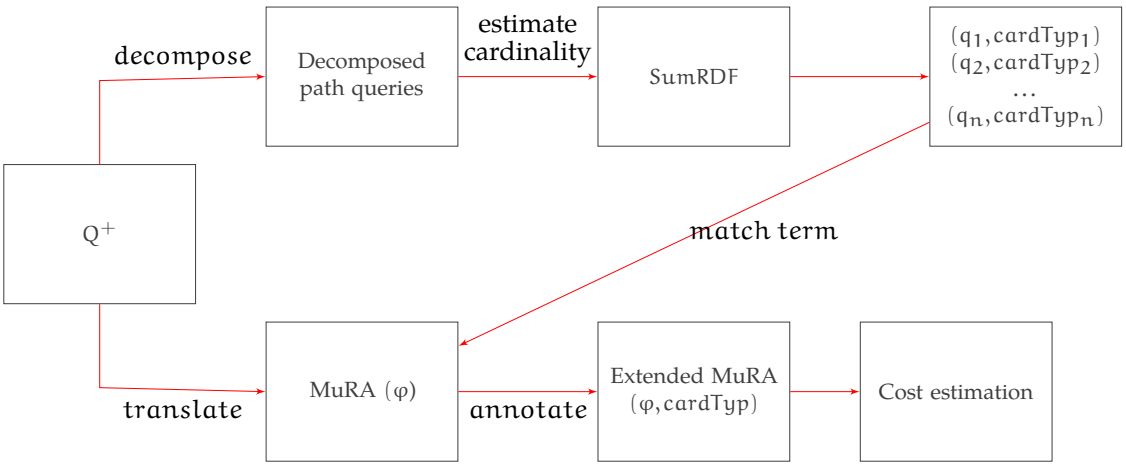


Figure 6.2 – SumRDF integration

Next, we retrieve the decomposed queries together with their respective cardinalities. The decomposed queries are matched with the LQEPs generated from the MuRA plan space and each subexpression or sub-term are annotated with their respective cardinalities. We refer to this new formed terms as the "Extended MuRA" which are simply a set of query plans with cardinality information.

In order to avoid estimation problems that could arise as a result of the query decomposition, we ensure that queries are decomposed and translated without loss of semantics and appropriate utilization the cost and cardinality estimation techniques discussed in Chapter 4. In addition, the matching phase ensures that decomposed subexpressions are correctly matched to the equivalent subexpressions of the query tree and that their cardinalities are updated respectively.

6.3.1 Query Decomposition Strategy

In the previous section we described the integration of SumRDF into our cost estimation framework. This section presents the algorithm for decomposing recursive regular path queries and match the decomposed subexpressions to nodes in the LQEPs. Following the decomposition algorithm, we examine the regular path queries constructs as described in [26]. We then proceed by giving examples and a detailed translation to SPARQL for the operators introduced in Chapter 2.

Algorithm 5 implements the cardinality estimation framework based on SumRDF [36]. It starts by decomposing complex queries into sub-queries on line 4. We then proceed by further decomposing any sub-queries that contains recursive queries *i.e.* fixpoints into *constant* and *recursive* parts with a call to the decomposition function on line 6. The decomposition function (lines 15 – 20) recursively decomposes sub-queries since queries in [26] can also contain nested fixpoint operators.

Let Q be a regular path expression with path p over a graph G such that $r_1, r_2, \ldots r_n \subset Q$, thus we define constructs of regular path expression in Table 6.1.

Algorithm 5 : RA term decomposition and cardinality estimation framework

Input: Recurive Path query in RA^+ form, Graph G, Summary S and target size t;

```
1:  card ← {};
2:  while (RA+ ≠ ∅) do
3:      if RA+ contain sub-structure then
4:          (q1, q2, ..., qn) ← DecomposeQuery(RA+);
5:          for k = 1 to n do
6:              subQuery ←DecomposeFX(qk)                    ▷ decompose fixpoint
7:              for each s in subQuery do
8:                  bgp[s] ← GetBgpEquiv(s)                  ▷ convert RA to SPARQL BGP
9:                  card ← estimateCard(bgp[s], G, S, t);    ▷ cardinality for each query
10:     else
11:         bgp[qn] ← GetBgpEquiv(qn)                        ▷ convert RA to SPARQL BGP
12:         card ← estimateCard(bgp[qn], G, S, t);           ▷ cardinality for each query
13: return card;

14: // fixpoint decompostion
15: function DecomposeFX(Q)
16:     if Q isFixpoint then
17:         (const, rec) ← SPLIT(Q);
18:         DecomposeFX(const);
19:     else
20:         result ← Q;
```

Table 6.1 – Path query definition

Construct	Description
r	edge
r_1/r_2	the concatenation of two paths
$r_1 \mid r_2$	the alternative choice between two paths
r^-	a reversed path
$r+$	the transitive closure of the path r

We now shift our focus to UCRPQ queries conversion to SumRDF SPARQL. We use Algorithm 5 for decomposing path queries into sub-paths for two reasons;

- to ensure that we adhere to the SPARQL fragment implemented in *SumRDF*, since they only support union the syntax is strictly compliant with SPARQL v1.0
- we are interested in the cardinality estimate for sub-paths in order to use the cost estimation equations of µ-RA.

To guarantee correctness of the decomposed query, a semantic-preserving translation is needed to ensure that the semantics of a subexpression or a term in the query plan

is equivalent to the semantics of the translated SPARQL query and that the expected cardinality is computed.

Definition 6.3.1. *(SPARQL query). We redefine a SPARQL query* Q_G *over a graph* G *as;*

$$Q_G \rightarrow [\![\text{SELECT } (v_1, v_2, \ldots v_n) \text{ WHERE } \{bgp\}]\!]_G \tag{6.2}$$

where $v_1, v_2, \ldots v_n \in var_i$ is an ordered list of query variables such that $var_i \subseteq var(bgp)$

We define a translation $trans$ function which translates a path query into semantically equivalent relational algebraic term. Table 6.2 summarizes the translation of path queries to relational algebra and their decomposition into SPARQL BGP.

Table 6.2 – Path query decomposition

	Translated RA form	**BGP**
r	$trans(r)$	$bgp(r)$
r^-	$trans(r^-)$	$bgp(r^-)$
r_1/r_2	$trans(r_1) \bowtie trans(r_2)$	$bgp(r_1 \,.\, r_2)$
$r_1 \vert r_2$	$trans(r_1) \cup trans(r_2)$	$bgp_1(r_1)$, $bgp_2(r_2)$
$r+$	$\mu\, X \,.\, trans(r) \cup trans(r/X)$	$bgp(r)$

As an example, a regular path query of the form r can be translated to SPARQL BGP in Table 6.3.

The cases described in Table 6.3 are simple cases and the translations are almost direct with no further decomposition is required. Path queries with alternative path of the form r_1/r_2 needs to be decomposed inside a single BGP block as $bgp(r_1\ r_2)$. While r_1 and r_2 are translated as Table 6.3 above. The decomposition of query of the form $r_1 \vert r_2$ yields two sets of SPARQL queries and $bgp_1(r_1)$ and $bgp_2(r_2)$ are automatically derived. Details of the fixpoint decomposition is given in the next section.

6.3.2 Fixpoint Decomposition

Fixpoint operator is of the form $r+$ and consists of the constant and the recursive part which is evaluated in 1 to several iterative steps. The relational algebraic form is given

Table 6.3 – BGP templates

Path	**Template**
r	$x\ r\ ?y$
$r-$	$?y\ r\ x$

below;

$$\mu X \cdot \ r \ \cup \ r/X$$

Before we dive into the decomposition phase, we first examine the query forms in SPARQL. There are two query forms; (i) queries with only query variables e.g. `?x r+ ?y` $=$ μ`X` $\cdot$`r` $\cup$ `X/r` (ii) queries with constant filter e.g. `c r+ ?x` $= \mu$`X` $\cdot \sigma_c($`r`$) \cup$ `X/r`.

At query translation time, each time there is a recursion of the form r+, there are two ways to translate it as a MuRA term: μ`X` $\cdot$r$\cup$`X/r` the one that navigates from left to right in the graph and μ`X` $\cdot$ `r`$\cup$`r/X` and the one that navigates from right to left in the graph. The two translations are obtained by using different renamings before joining `X/r` which stands for `join(rename(X), rename(r))`. This is the reason why a set of MuRA terms is obtained from any given RPQ/CRPQ/UCRPQ. The two translations (corresponding to both directions) are equivalent so they end up having the same cardinality estimation.

However whenever filters are involved (constants used in CRPQs introduce filters), then the two navigations might have a very different cost. Filters introduced after algebraic term transformations (which causes an increase in the number of MuRA terms). After this phase, filters may have been pushed inside the constant part (and/or even the recursive part) of a fixpoint. The different ways of navigating may have a significantly different costs. This cost heavily depends on the direction of navigation and on the size of intermediate results, and in particular the number of the initial graph nodes from which the navigation starts. Hence, having accurate cardinality estimations is instrumental to precisely estimate a cost which is used for selecting which algebraic term variants will be passed to the next phase; the physical plan selection. This is why it is important to determine these costs by SumRDF cardinality estimation. Now let us examine at when we can achieve that properly.

At translation times, (if we keep our current way of translating), filters basically stay out of recursions. That is, if we have a constant like in `c r+ ?x` then you will end up having $\sigma_c(\mu X \cdot$ r$\cup$X/r$)$ and $\sigma_c(\mu$`X` $\cdot$ `r`$\cup$`r/X`$)$ which is not convenient for using SumRDF estimation at this stage. After applying transformation rules, however, the filters will be pushed inside the mu (whenever possible), and this will yield (among other terms): μ`X` $\cdot \sigma_c($r$) \cup$ `X/r` and μX.$\sigma_c($r$) \cup$ r/X. This is getting better because at this stage, it becomes easier to resort to SumRDF cardinality estimations to select which one is the best.

We use Algorithm 5 to precisely do the decomposition and translation. We then retrieve the cardinality for the decomposed queries from SumRDF.

6.3.3 Annotating Query Plans with Cardinality Estimate

After retrieving the cardinality estimate for queries in SumRDF, the next task is to annotate the relational algebraic tree (i.e. logical plan) with these cardinality estimates. In a first step, we extend the definition of the relational algebraic framework presented in MuRA [26] with cardinality type `cardTyp` which is the cardinality retrieved from SumRDF in this case. The grammar for the extend RA framework is presented below.

$$
\begin{array}{lr}
\varphi^{\tau} ::= & \text{term (with cardinality)} \\
\quad (\mathsf{E}, \mathsf{cardTyp}) & \text{relation variable} \\
\quad (\mathsf{X}, \mathsf{cardTyp}) & \text{recursive variable} \\
\mid \mathsf{const}(\mathsf{c}, \nu, \mathsf{cardTyp}) & \text{constant} \\
\mid \mathsf{union}(\varphi_1, \varphi_2, \mathsf{cardTyp}) & \text{union} \\
\mid \mathsf{join}(\varphi_1, \varphi_2, \mathsf{cardTyp}) & \text{join} \\
\mid \mathsf{antijoin}(\varphi_1 \triangleright \varphi_2, \mathsf{cardTyp}) & \text{antijoin} \\
\mid \mathsf{filter}(\varphi, \mathsf{pred}, \mathsf{cardTyp}) & \text{filtering} \\
\mid \mathsf{rename}(\varphi, \mathsf{a}, \mathsf{b}, \mathsf{cardTyp}) & \text{renaming} \\
\mid \mathsf{remove}(\varphi, \mathsf{a}, \mathsf{cardTyp}) & \text{anti}-\text{projection} \\
\mid \mathsf{decMu}(\varphi, \mathsf{X}, \mathsf{R}, \mathsf{cardTyp}) & \text{fixpoint}
\end{array}
$$

Grammar: Extended MuRA grammar

6.4 Evaluation

In this section, we present the evaluation of our cost estimation technique by comparing it against state-of-the-art systems. Our results are categorized into three parts; in the first part, we present the cardinality and selectivity evaluation coupled with the varying effects of data statistics calibration on query performance. Secondly, we demonstrate the effectiveness of our recursive cost estimation framework using recursive UCRPQ queries over graph datasets of varying sizes.

6.4.1 Experimental Setup

We conduct several experiments to assess the effectiveness of our cost estimation technique by evaluating recursive graph queries (that are union of conjunctive regular path queries [103]) using the translation found in [26] for the recursive relational algebra. A graph query is first translated into a term φ, then all equivalent terms are exhaustively enumerated resulting in a plan space $\mathcal{P}$. Finally among all terms in $\mathcal{P}$, the term φ which minimizes a cost estimation function f is retained, and executed.

We evaluate queries using two systems corresponding to two settings:

- *System P* is the popular PostgreSQL system [104], where f is the function that returns the cost estimated by PostgreSQL using the explain API;
- *System P'* is also the PostgreSQL system, but where f is the cost function that we propose in this thesis.

The comparison between the two settings is fair since the only difference is the cost estimation function.

Experiments on the Yago dataset were conducted on a 128 GB RAM server with *2 Intel Xeon E5-2630 v4 CPUs (2.20 GHz, 20 cores each)*. All other experiments reported here were conducted on a 16GB RAM *Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz* machine.

6.4.1.1 Datasets

We experiment with two kinds of datasets: a real-world dataset: Yago2s [105] and two synthetic datasets (Shop and Uniprot) generated by the *GMark* system [106]:

Table 6.4 – Dataset and statistics

		Column cardinality		
Dataset	cardinality	src	trg	predicate
Yago2s	62,643,951	35,165,791	8,572,450	83
Shop	268054	46,696	88,454	81
Uniprot	773,280	86,000	11,8124	7

* column cardinality refers to the number of distinct values in each column

Table 6.4 shows the data statistics for data used in this evaluation. In some cases, we vary the number of nodes in these data especially the synthetic (generated) ones.

6.4.1.2 Queries

We experiment with a total of 40 regular path queries specifically UCRPQ, CRPQ and RPQ queries. These queries have at least one recursive path.

6.4.1.3 Metrics

We measured different performance metrics for each the queries against each dataset and system by profering answers to the following question.

- How accurate are the statistics and what is their impact on the query time?
- What is the impact of (1) cardinality estimation and (2) cost model on query plan quality?
- How does our approach compare to the state-of-art in terms of query runtime?

The questions above led to the introduction of metrics such as query running time, number of iterative steps in a fixpoint, size of join and their effect on query plan quality as well as the effect of the different cardinality and selectivity estimation techniques considered in this work. Other metrics considered include plan cost estimation time, ranking cost model in quartiles as disccused in the previous chapter and [20].

6.4.2 Data Statistics

We compare the statistics generation time for three (3) different datasets. This experiment is carried out to measure the time taken to compute the data statistics for each of the datasets considered. Computing the exact statistics (e.g. counting the number of tuples in the dataset) is slower since each row or node needs to be accounted for. As a result we rely on the estimated values for the statistics but when unavailable we compute the exact statistics.

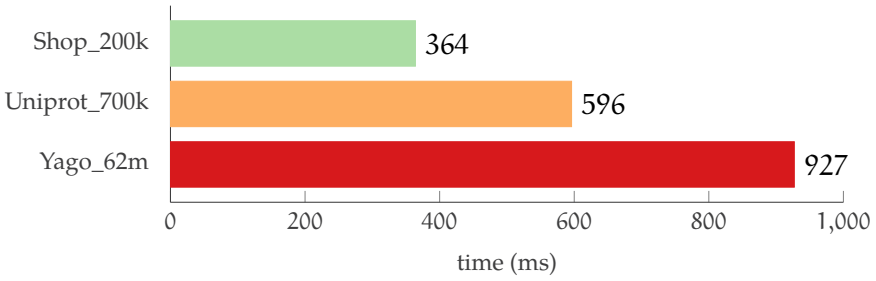


Figure 6.3 – Statistics computation time

Figure 6.3 shows the time taken to compute the data statistics. Yago dataset has the highest statistics computation time followed by Uniprot and Shop datasets. The label consist of the graph name together with the number of nodes present in the dataset. The higher the number of nodes is, the more time it takes to compute the statistics.

6.4.3 Cardinality Estimation

In Chapter 4, we defined cardinality estimation and a set of formulas for estimating the cardinality of the different operators we have identified. In this section, we examine the quality of our cardinality estimation framework and the formulas described earlier through;

- the precision or closeness to reality of our cardinality estimation for recursive queries
- estimation error and their effect on the overall plan quality

Estimating the cardinality for recursive queries is more complex compared to non-recursive ones. The evaluation of recursive queries happens in several steps and this makes it even harder to project the final estimate of the cardinality of the fixpoint operator. This experiment give insights into how precise the cardinality estimation formulas are for recursive queries. RA operators considered here include, renaming, filter and anti-projection. We consider these operator not independently but combined in a full query plan through intermediate results.

With this experiment, we compared the estimated cardinality with the exact cardinality i.e. the actual result count after executing the query against the datasets. For the purpose of the experiment in this section, we consider only non-empty queries.

Following the work of [102] and as discussed in Chapter 5, we use the q-error (in Equation 6.3) to determine the factor by which the cost model deviates from the actual cardinality.

$$\text{q-error}(Q) = \max\left(\frac{c}{\hat{c}}, \frac{\hat{c}}{c}\right) \tag{6.3}$$

For any given query Q, we consider the q-error which measures in absolute the deviation of estimated cardinality (c) from true cardinality ($\hat{c}$).

We also use the estimation error $s'(Q)$ defined in Chapter 5.

$$s'(Q) = \begin{cases} \frac{\hat{c}-c}{\hat{c}}, & \text{if } c > \hat{c} \\ \frac{c-\hat{c}}{c} & \text{otherwise} \end{cases} \tag{6.4}$$

Table 6.5 – Cardinality Estimation for queries

#	$c(Q)$	$\hat{c}(Q)$	q-error(Q)	$s'(Q)$
Q_1	32	20	1.6	0.6
Q_2	108	250	2.31	-1.31
Q_3	7	89	12.71	-11.71
Q_4	6	36	6	-5
Q_5	148	301	2.03	-1.03
Q_6	128	118	1.08	0.07
Q_7	1	465770	465770	-465769
Q_8	8	4909	613.6	-612
Q_9	10	4	2.5	0.4
Q_{10}	51	3	17	0.94
Q_{11}	6	3	2	0.5
Q_{12}	4563	1	4563	0.99
Q_{13}	190	3	63.33	0.98
Q_{14}	351	3	117	0.99
Q_{15}	48	3	16	0.94
Q_{16}	264721	3	88240	0.99
Q_{17}	24118	3	8039.3	0.99
Q_{18}	46	4	11.5	0.91

Table 6.5 shows the actual and estimated cardinality, the q-error and estimation error factor for selected queries across three (3) different datasets.

The q-error(Q) column shows the cardinality estimation error from the cost model and as we see in the table the cost model estimates the cardinality with at least an error factor of two in most cases. The fifth column in the table shows the estimation error factor $s'(Q)$. In queries $2,3,4,5,7,8$, the cost model overestimates the cardinality of the queries and underestimates the cardinality for all other cases.

To give more insights to how accurate the current cardinality estimation method is, we group the q-error into intervals in Table 6.6 (according to the format in Chapter 5) with $[0,1]$ meaning the estimation is 100% accurate and the inaccuracy grows as we move down the table.

Table 6.6 – q-error interval for cardinality estimations

q-error interval	Frequency	Percentage of total queries
[0, 1]	-	0
[1, 2]	3	16.7
[2, 4]	3	16.7
[4, 8]	1	5.5
[8, 17]	4	22.2
[17, 10^3]	3	16.7
[10^3, 10^5]	4	22.2

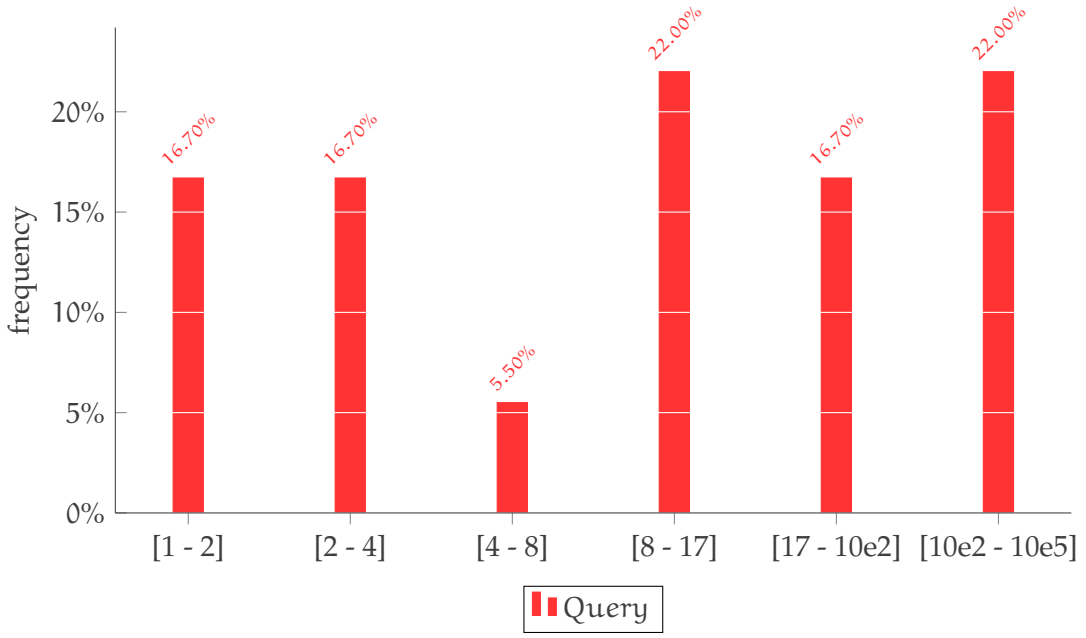


Figure 6.4 – Distribution of queries on q-error interval

As shown in Figure 6.4, the cost model does not accurately estimate the cardinality for any of the queries considered. In the $[1,2]$ interval, 16.7% of the queries is misestimated by a factor of at most 2. Another 16.7% falls within the $[2,4]$ q-error interval. The largest misestimation comes in the $[8-17]$ and $[10^3, 10^5]$ range.

We investigate next, the effect of cardinality misestimation on the overall query execution time.

Tracking Result Size in Fixpoint Operator

Recall that in Chapter 4, we present the *step-wise* cardinality where we present the formula that reflects the changes in result size of the recursive operation of a fixpoint operator. We seek here to validate the formula given in Equation 4.43 by analyzing and tracking the increase in the result size of the recursive part of the fixpoint operator. Similar to Equation 6.3, the size increase can be estimated by taking the final result size at the end of the recursive computation minus the cardinality at the first step of the recursive computation (see Table 4.6).

$$\Delta_{rec} = rowCount_{\mathbb{N}} - rowCount_{step_0} \tag{6.5}$$

where Δ_{rec} is the incremental factor, $rowCount_{step_0}$ is the number of tuples at the beginning of the iteration and $rowCount_{\mathbb{N}}$ is the result size at the final step $\mathbb{N}$ of the iteration just before combining them with the result of the constant part of the fixpoint operation.

Table 6.7 – Result size for queries on Uniprot graph

#	$\mathbb{N}$	$step = 0$	$step = 1$	$step = \mathbb{N}$	Δ_{rec}
Fx_1	7	53	3083	3141	3088
Fx_2	9	55	365202	371906	371851
Fx_3	7	15	5571	5931	5916
Fx_4	12	48	1.33×10^8	1.36×10^8	1.359×10^8
Fx_5	6	19	351	370	351
Fx_6	9	33	836710	862612	862579
Fx_7	8	101	31477	31790	31689
Fx_8	8	84	17154	17360	17276
Fx_9	11	137	1.66×10^7	1.66×10^7	1.659×10^7
Fx_{10}	10	33	1641552	1692169	1692136

If we take Fx_1 in Table 6.7 for example, the number of tuples in the beginning of the iteration is 7. At maximum iteration $\mathbb{N} = 7$, the intermediate result size of the recursive part of this fixpoint is 3141 *i.e* after 7 iterations no new results were be added to the intermediate results. At this point the intermediate result size is combined using a union operation.

6.4.3.1 SumRDF Cardinality Estimation

We decomposed queries on the three different datasets and performed cardinality estimation of those decompositions as described earlier. We quantify the accuracy and efficiency of SumRDF estimation technique using an upperbound q-error [102].

There are 40 queries in total, 20 of which are evaluated on synthetic graphs generated using GMark and the other 20 queries are evaluated on Yago2s graph. We considered

Table 6.8 – Graph, summary for SumRDF estimation

Dataset	Graph G		Summary S				Queries	
	#resources	#triples	buckets	triples	reduction factor	constr. time	Q_0	Q_{dec}
Uniprot	92 k	382.8k	2k	151	2535.5	0.56 s	10	361
Shop	57.1k	93.3k	2k	36.9k	2.53	0.83 s	10	789
Yago2s	42.8 M	62.5 M	20k	1.9 M	22.48	15.9 h	20	173

* Q_0 is the initial number of queries
* Q_{dec} is the number of decomposed queries

every combinations of the decomposed queries so that every posible combination of subexpressions on the query tree can be accounted for. We use a Uniprot graph consisting of 382.8k triples with 92k resources which when summarized using 2000 buckets shrink the original graph G to a summary S of 151 triples. Similarly, the Shop dataset is summarized using the same number of buckets and yields 36.9k summaries. Yago2s graph, being the largest contains 62M triples, 42.8M resources with a summary triple of 1.9M taking a little more than 15h to construct the summary.

Table 6.9 – q-error interval for SumRDF estimations

q-error interval	Frequency		
	Uniprot	Shop	Yago2s
[0, 1]	209	598	166
[1, 2]	101	180	7
[2, 4]	17	11	-
[4, 8]	18	-	-
[8, 45]	16	-	-

In Table 6.9, we use intervals to capture the q-error for all queries on the three considered datasets. The interval $[0, 1]$ represents the case where there is no error i.e. the number of tuples returned by SumRDF is the same as the actual number of tuples. An error bound of $[1-2]$ represents the case where the estimated cardinality differs from the true cardinality by at most a factor of 2. Generally, the more values a dataset has in the lowest bound, the higher the accuracy of the estimation technique for that dataset is.

Figure 6.5 shows the q-error for the three datasets. In 57% of the queries for Uniprot graph, the SumRDF estimation technique accurately predicts the cardinality while 27% of the estimation lies in the $[1-2]$ bound. Another 4.9% and 4.3% are erroneously predicted having q-error in $[4-8]$ and $[8-50]$ respectively. Estimation for Shop queries resulted in 75% accuracy with 22% in the $[1-2]$ and 1.4% in the $[4-8]$ q-error range respectively. Cardinality estimation of the queries on Yago2s yields a little above 95% accuracy having just 4.04% in the $[1-2]$ error interval.

We have seen so far how SumRDF cardinality estimation works on three datasets. In the next chapter, we will show how cardinality estimation helps guide the cost model for selecting accurate plans.

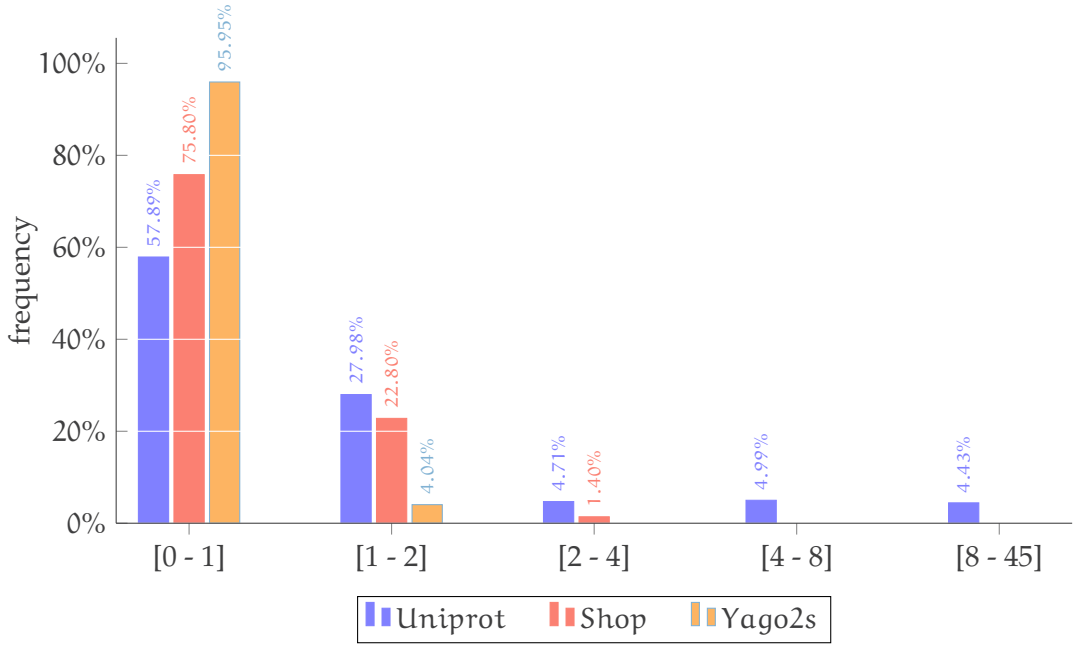


Figure 6.5 – Distribution of queries on q-error interval

6.4.4 Relative Query Performance

We evaluate the cost model for recursive queries by assessing the plan quality of the query plans selected by the cost model in terms of the query runtime, cost, cost evaluation time and the average query time for queries on the different datasets. Query performance is given in terms of the actual query time.

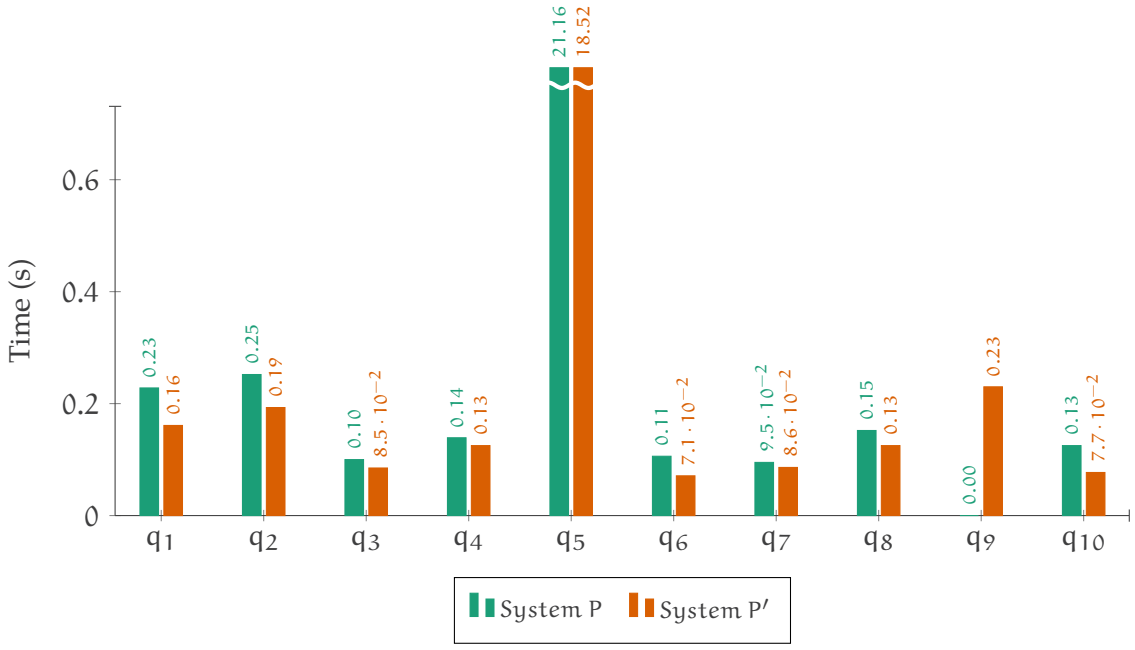


Figure 6.6 – Query evaluation times for queries on the Uniprot datasets

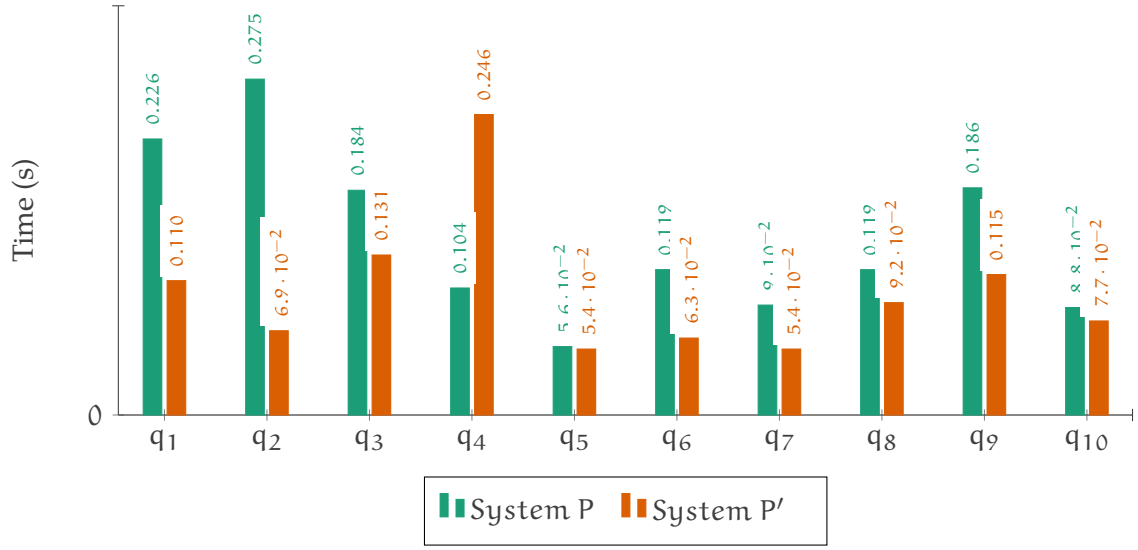


Figure 6.7 – Query evaluation times for queries on the Shop datasets

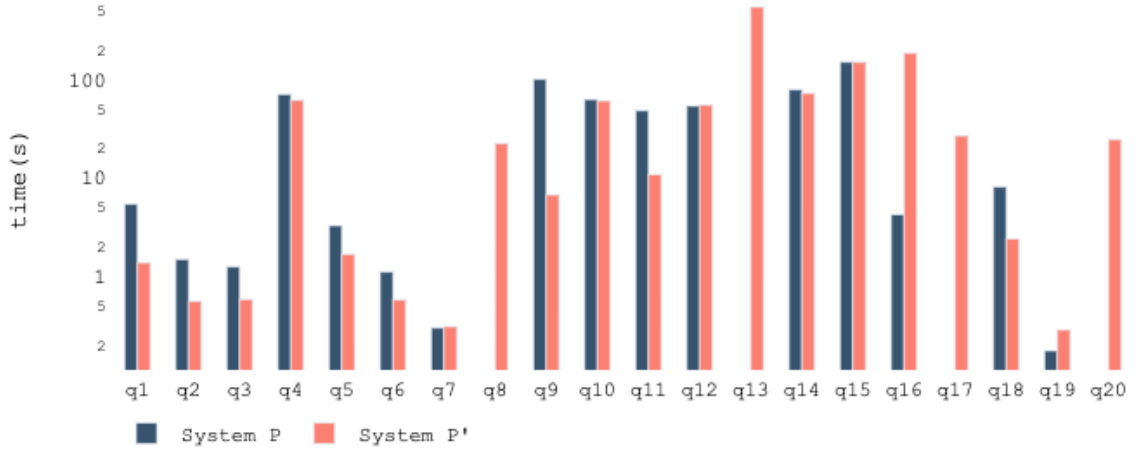


Figure 6.8 – Query evaluation times for queries on the Yago2s dataset

Figure 6.6 shows the query evaluation time for 10 queries from GMark Uniprot graph. Results shows that we outperform System P in all cases considered. In particular, we outperform SystemP in query 5 by 2.64 seconds and by a few milliseconds in all other cases. SystemP timeout after a period of 30 minutes in query 9.

Figure 6.7 shows the times spent by the two systems for evaluating queries on the GMark Shop datasets. Results show that both systems evaluate all the queries in a comparable amount of time: all queries are evaluated by both systems in less than 0.3 seconds. Specifically, *System P′* outperforms *System P* for 9 out of the 10 queries. For the remaining case of q_4 on the Shop dataset, *System P* performs better by 150 milliseconds.

Figure 6.8 shows the times spent for evaluating the 20 queries of [26] on the real-world Yago2s dataset [105]. Results show that *System P′* outperforms *System P*. In particular, System P could not answer queries $q_8, q_{13}, q_{17}, q_{20}$ within the allowed time frame of 15 minutes, while System P′ evaluates these queries in 146, 108, 34 and 14 seconds,

respectively. For the 16 other queries remaining, the results of System P′ are comparable or even slightly better than System P′.

6.4.5 Simplified (Cmm) Cost Model

We experiment with the Cmm cost model from Leis et al.[55]; a simplified cost model designed to serve as a baseline for more complex cost models. This cost function put emphasis on the cardinality of the join and it is only counts the number of tuple that pass through each operator.

$$C_{mm}(T) = \begin{cases} \tau \cdot |R|, & \text{if } T = R \vee T = \sigma(R) \\ |T| + C_{mm}(T_1), + C_{mm}(T_2), & \text{if } T = T_1 \bowtie T_2 \end{cases} \tag{6.6}$$

In addition to not modeling the I/O cost, the Cmm cost model brings about changes to the way we calculate the cost for join and filter operators such that a constant $\tau = 0.2$ is now used in place of filter selectivity factor which is calculated in our cost model. The general formula for calculating the cost of recursive operator still remains the same as the one presented in Chapter 4 because the Cmm cost model does not give any cost formula for recursive operator. R represents the base relation, T_1 and T_2 are the left and right relations respectively. T is the join result size.

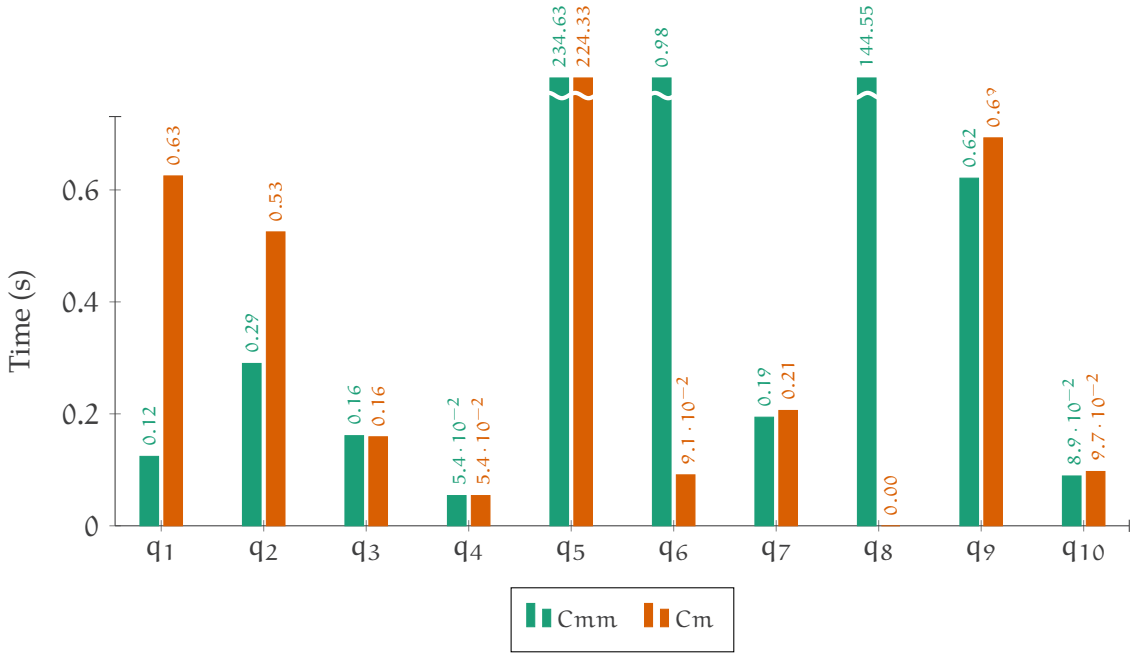


Figure 6.9 – Comparison of the execution times for Cmm and Cm on Uniprot datasets

The result from our experiment with Cmm cost model are shown in Figure 6.9. Cm represent the cost model proposed in this work i.e. System P′. The simplified cost model performs significantly better in Q_1, Q_2 and has relatively the same performance as Cm in Q_7, Q_9 and Q_{10}. In query 5, however, Cmm cost model perform poorly by 10 seconds.

Table 6.10 – Cost and runtime for different cost models

	Cm Runtime (s)	Cm Est. Cost	Cmm Runime (s)	Cmm Est. Cost
Q_1	0.625	6613	0.124	1.03×10^{17}
Q_2	0.525	9518	0.272	8.85×10^{12}
Q_3	0.159	4640	0.161	6.07×10^{14}
Q_4	0.054	230947	0.054	2.40×10^{17}
Q_5	224.329	902535869	234.628	9.89×10^{10}
Q_6	0.091	5514818712	0.98	5.96×10^{14}
Q_7	0.206	3755	0.194	1.752×10^{12}
Q_8	-	5990180	144.545	4.02×10^{15}
Q_9	0.693	50954	0.621	9.37×10^{17}
Q_{10}	0.097	6932	0.089	1.01×10^{12}
Q_{11}	0.094	6519818	0.226	4.95×10^{10}
Q_{12}	1.746	78262248	4345.940	7.44×10^{13}
Q_{13}	0.193	9256831	0.470	2.39×10^{15}
Q_{14}	0.413	369475	0.590	9.24×10^{14}
Q_{15}	0.098	27322	0.082	4.24×10^{10}
Q_{16}	0.117	5343762	0.179	1.55×10^{12}
Q_{17}	0.155	5512520	0.145	2.75×10^{13}
Q_{18}	0.277	18700	0.173	6.22×10^{12}
Q_{19}	0.634	16366	0.712	4.45×10^{11}
Q_{20}	0.172	4665	0.121	4.31×10^{9}
Avg. time	11.5s		241.3s	

* Cm default cost model (i.e. System P′ cost model)
* Cmm simplified cost model

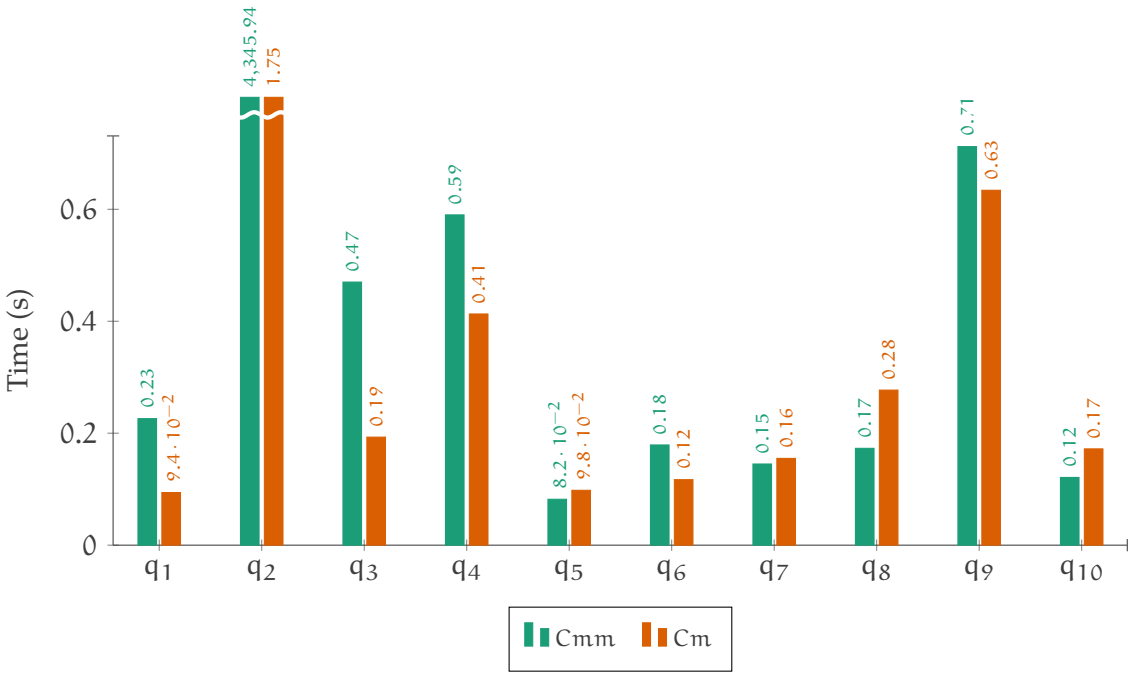


Figure 6.10 – Comparison of the execution times for Cmm and Cm on Shop datasets

C_m cost model time out after 27 minutes and does not return any result. In all other cases, C_m cost model is better but compared to the simplified cost model C_{mm}, the overall performance of the simplified model is better for this case.

Figure 6.10 shows the execution times for queries on Shop graph using the C_{mm} and C_m (our default cost model). Results shows that we outperform the simplified cost model in six cases out of 10. In particular, the simplified cost model C_{mm} perform worse by an order of magnitude in queries 1 and 2 and slightly worse in queries 3, 6 and 9. For the rest of the cases, our cost model runs in almost comparable times where as in queries 5, 7, 8, and 10 it performs slightly better than the default cost model.

From the results, we can deduce that using our cost formula for recursive operator even with a simplified cost model that only count the number of tuple that passes through each operator gives a comparable performance. Q12 is an exception where the simplified cost model selects a query plan with very bad performance making the average query runtime (for all of the 20 queries) of the simplified model to be $\times 21$ more than our default cost model.

6.4.6 Impact of True Cardinality on Plan Quality

To understand the impact of accurate cardinality estimation on the quality of the plan selected by the cost model, we compare the time spent evaluating queries selected by our cost model without accurate cardinality estimation and the time spent evaluating queries selected by our cost model using the SumRDF cardinality estimation technique. We use $\mathrm{System\ P'} + \mathrm{SumRDF}$ to represent when we use SumRDF estimated cardinality on our cost model.

In Figure 6.11, we compare the time spent evaluating the queries on SystemP, $\mathrm{System\ P'}$ and by using the true cardinality with $\mathrm{System\ P'}$ ($\mathrm{System\ P'} + \mathrm{SumRDF}$) on Uniprot graph. Results shows that $\mathrm{System\ P'} + \mathrm{SumRDF}$ outperforms $\mathrm{System\ P}$ in 8 out of 10 cases and $\mathrm{System\ P'}$ (where we plugged the real cardinality into our cost function) in 7 out of 10 cases. For q_3 and q_10, $\mathrm{System\ P}$ performs better by 1.93 seconds and 13 milliseconds respectively.

Similarly, Figure 6.12 shows the results of evaluation of 10 queries on the Shop graph. Results also shows that $\mathrm{System\ P'} + \mathrm{SumRDF}$ outperforms $\mathrm{System\ P}$ in 9 out of 10 cases considered and outperforms $\mathrm{System\ P'}$ in all 10 cases considered. Only in q_1 that $\mathrm{System\ P}$ was better than our cost function with the true cardinality from SumRDF estimation by 150 milliseconds.

Figure 6.11 and Figure 6.12 shows that having accurate cardinality estimation improves the quality of the query plan chosen by the cost model. In contrast, Figure 6.13 shows that this is not always the case. Results show that the systems under consideration evaluates most of the queries in a comparable amount of time and having accurate cardinality does not significantly change the result outcome when we rely on the estimates from our cost model. When we plug the cardinality estimate from SumRDF ($\mathrm{SystemP'} + \mathrm{SumRDF}$),

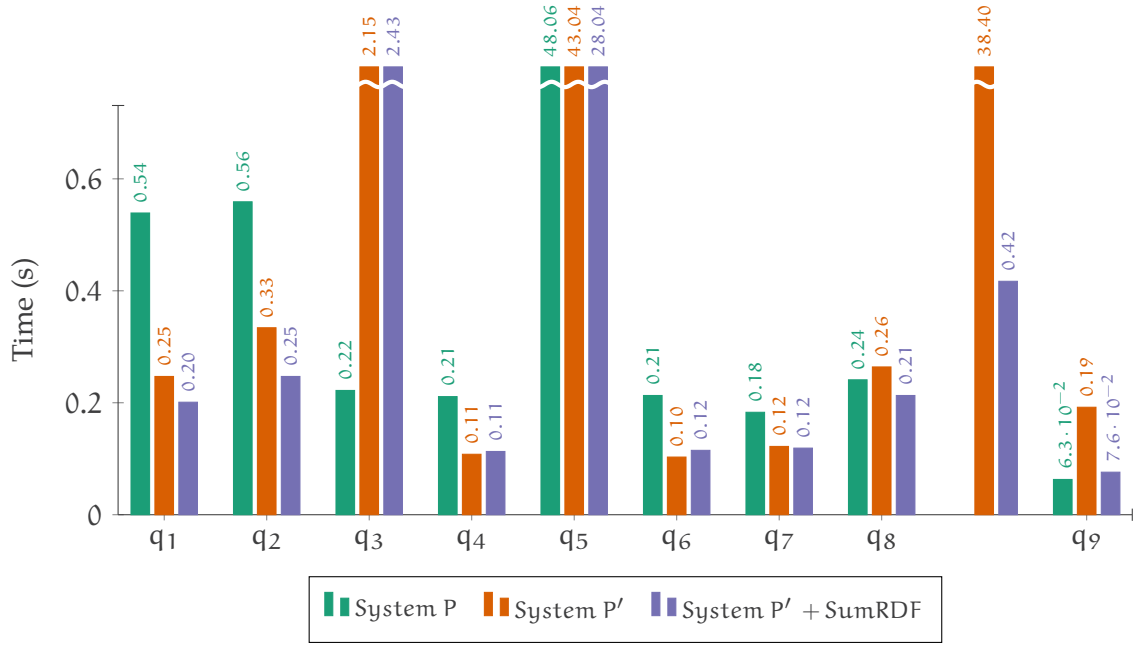


Figure 6.11 – Query evaluation times using SumRDF technique for queries on the Uniprot datasets

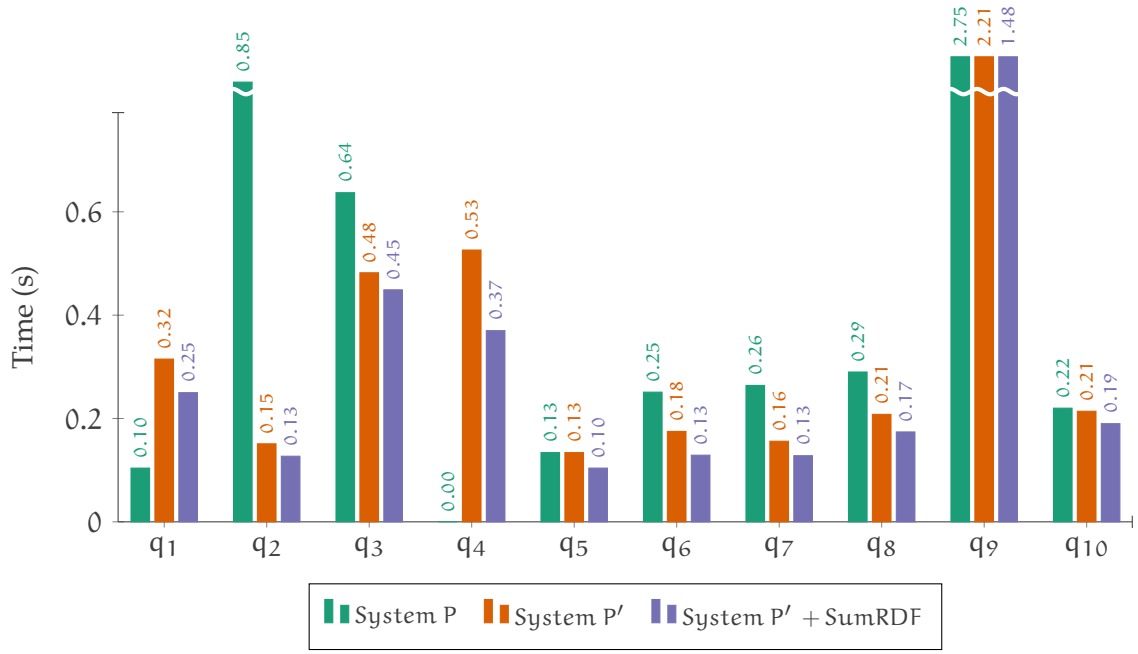


Figure 6.12 – Query evaluation times using SumRDF technique for queries on the Shop datasets

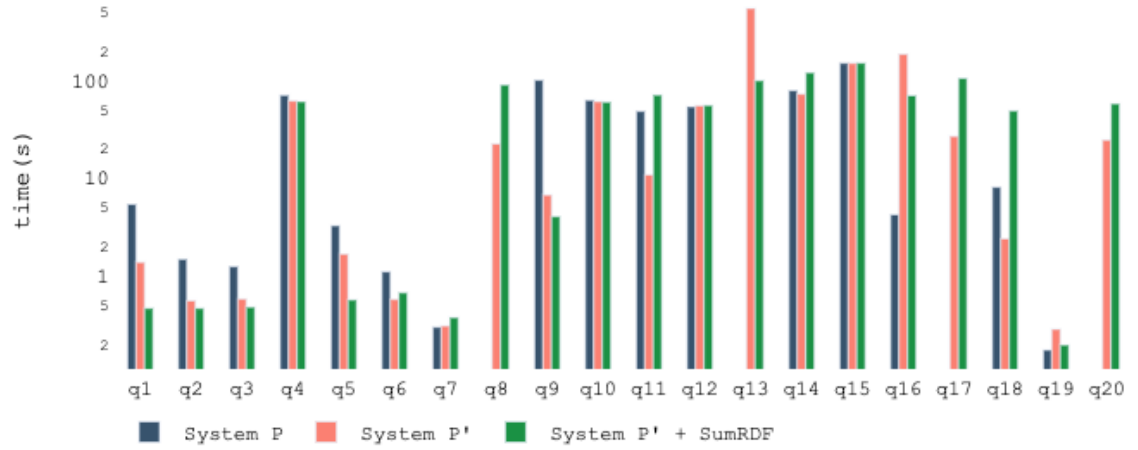


Figure 6.13 – Query evaluation times using SumRDF technique for queries on the Yago2s datasets

the query evaluation times in 8 cases; $q_1 - q_5$, q_9, q_{10}, and q_{13} are better than the rest. Surprisingly, the misestimated cardinality from our cost model ($SystemP'$) yields better performance in seven cases; $q_6, q_{11}, q_{14}, q_{15}, q_{17}, q_{18}$, and q_{20}. A few exceptions show that bad cardinality estimation leads to very poor performance as in q_1, q_{13} and q_{16}.

This illustrates the practical usefulness of the refined cost estimation presented in this work.

6.4.7 Cost Estimation Time vs. Number of Plans

Using the two generated datasets, we evaluate the cost estimation time and the number of plans generated for each of the considered 20 queries. The cost estimation time in this context refers to the time spent computing the cost for all the plans generated from the plan space for a given query.

Table 6.11 – Query execution details for queries on Shop dataset

#	Plan cost time (ms)	Query time (ms)	#Plans
Q1	1022	402	10110
Q2	1051	-	176683
Q3	1696	600	348826
Q4	991	238	119399
Q5	832	120	95363
Q6	1178	259	247279
Q7	1506	7417	599497
Q8	1143	138	230809
Q9	659	7854	29310
Q10	1219	350	210952

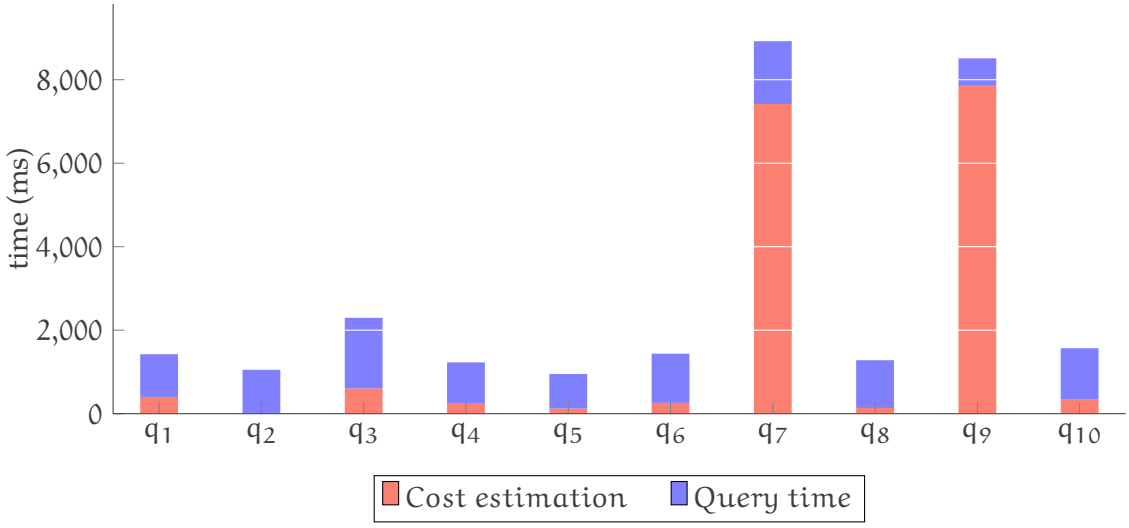


Figure 6.14 – Plan cost computation and query times for Shop queries

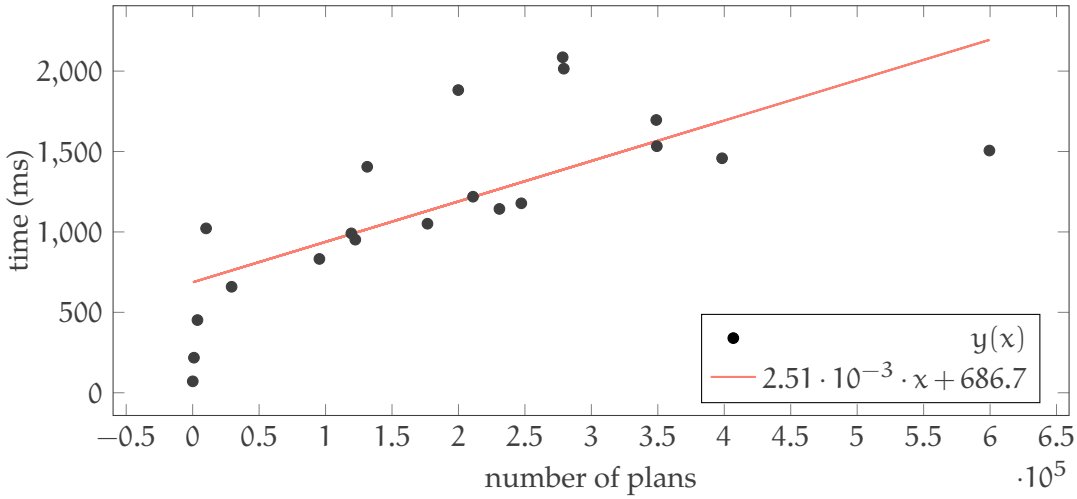


Figure 6.15 – Correlation between cost estimation time vs. number of plans

Figure 6.14 shows the time spent computing the cost for all of the equivalent plans generated in the plan space for each query and the time spent evaluating each plan selected by the cost model for each query. In q_7, the time spent costing the query plans is significantly higher since the number of plans is high. For q_9, the cost estimation is unsually high (higher than q_7) and the number of query plans costed is lower than q_7. A factor that contributed to this is the complexity of the query considered. In all other cases, the time spent evaluating the query are higher than the time spent costing the query plans.

Furthermore, we are interested in understanding (i) the correlation between the cost estimation time and the number of plans (ii) predicting the amount of time spent computing the cost for a single query.

Figure 6.15 shows the scatterplot of the cost estimation time and number of query

plans. We observe a strong positive correlation (with coefficient $r = 0.71$) between the cost estimation time and the number of plans generated. According to the model, each additional query plan is associated with a cost of $2.51 * 10^{-3}$ millisecond.

6.4.8 Varying Graph Size

We performed additional experiments with a varying number of nodes with an intention to understand how increasing graph size can affect the query evaluation time. Using the GMark [106] synthetic dataset, we are able to generate graphs of varying size. The experiment was carried out using the same set of queries over different graph size.

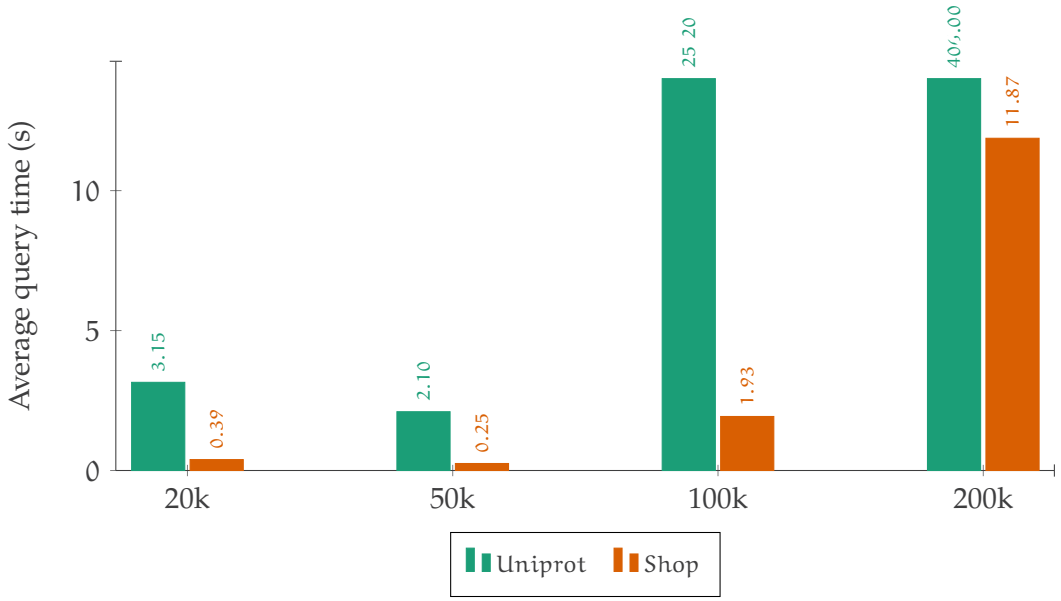


Figure 6.16 – Average query time for queries on GMark graphs

Figure 6.16 shows the result of executing the same queries over 20k, 50k, 100k and 200k nodes for Uniprot and Shop graph datasets. Result shows that the smallest graph of 20k nodes have higher average query time (3.15s and 0.39s respectively) than the graph of 50k nodes. From 50k upward, we see a trends where increasing the graph size also increases average query time.

6.4.9 Ranking of Cost Estimations

We also run all equivalent terms of the plan space P that are generated by the optimizer, in order to assess how our term-picking function compares to the best terms of P: the ones with the minimum actual measured query times. Cm represents the cost function presented in this work, Cmm is the simplified cost model [55] disccused earlier in this chapter and PostgreSQL is the cost function for PostgreSQL.

Figure 6.17 shows the number of queries for which the plan picked by each system are the ones with minimum cost, or in the 15^{th} percentile, the 25^{th} percentile, etc. among all

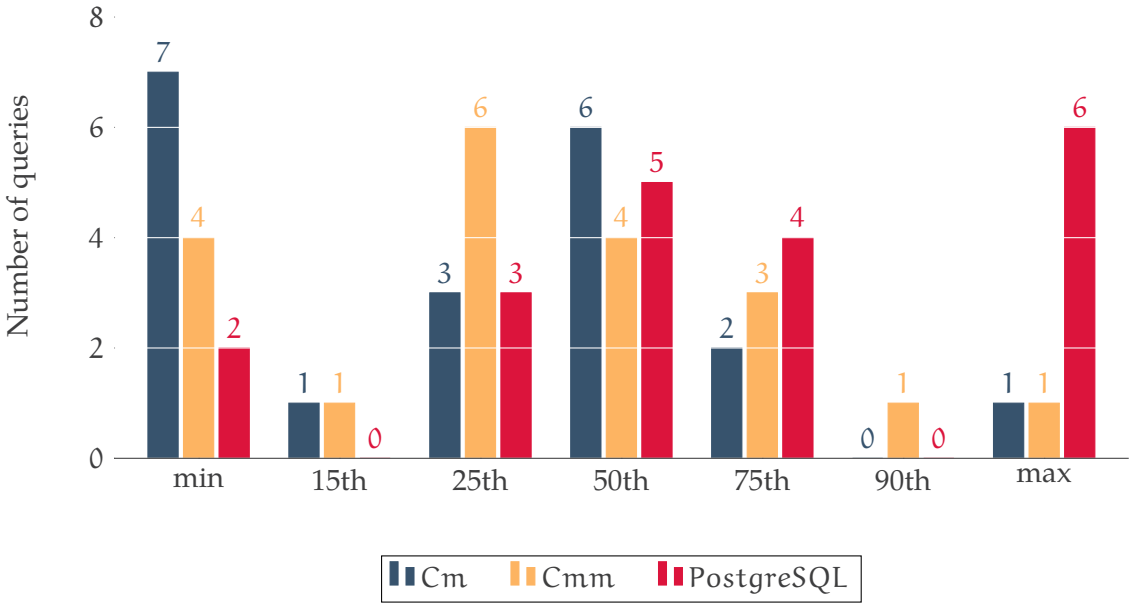


Figure 6.17 – Query rank by percentile for GMark

plans in P ranked in increasing order of actual running times. We observe that our cost model Cm picks more efficient terms more often. In 7 out of the 20 queries, our cost model selected the QEPs with the minimum evaluation time compared to 4 for the simplified model and 2 for PostgreSQL. We can see that most queries plans selected by PostgreSQL cost model mostly falls above the 25^{th} percentile (50^{th} percentile and above) and QEP with the slowest evaluation time in 6 cases. Our cost model and the simplified cost model both selects one plan each which ranked in the 100^{th} (max) percentile.

Table 6.12 – Ranking for cost model

Cost model	#Queries	Rank score
Cm	8	1^{st}
Cmm	5	2^{nd}
PostgreSQL	2	3^{rd}

If we use the 15^{th} percentile as our baseline for the ranking, we can count the number of queries that satisfy the condition presented in Chapter 5. We give the ranking score in Table 6.12. This ranking also illustrates the effectiveness of our estimation techniques.

6.4.10 Accuracy of Cost Estimation

In order to validate the effectiveness of our cost model Cm, calculate the estimation error (see Chapter 5) generated, comparing this error with the ones generated by the Postgres cost model and Cmm the cost model discussed earlier in this work. We compare the estimation errors for 20 queries on the synthetic datasets from GMark [106]. For each

query considered, we measure the running time for all possible query plans. Because we need to execute all the equivalent plans generated by the query optimizer in order to get their actual execution time, we reduce the number of nodes for each graph to just 100k nodes.

Using the formula for relative error defined in Equation 5.10, we can substitute the cost for the selected plan for the actual query evaluation time as follows;

$$Err = \frac{T_P - T_{min}}{T_{min}}$$

where T_P denotes the query evaluation time for the query plan selected by the cost model and T_{min} represents the plan with the minimum query evaluation time in the plan space. The closer the relative error is to zero the more effective the cost model is.

Table 6.13 – Cost estimation error

	Query	Err_{Cm}	Err_{Cmm}	$Err_{postgres}$
Uniprot	Q1	0	0.38	0.608
	Q2	0.0516	0.0516	0.0516
	Q3	0.146	0.662	0.131
	Q4	0	0.601	0
	Q5	0.507	0.507	0
	Q6	0.0375	0	0.169
	Q7	0.0631	0.0631	0.0842
	Q8	0	0	0.385
	Q9	0.757	0.757	0.757
	Q10	0	1.255	1.704
Shop	Q11	0.0204	0.0204	0.0204
	Q12	0.022	0.022	0.044
	Q13	0.097	0.097	0.097
	Q14	0.125	0.1607	0.25
	Q15	0	0	0.03
	Q16	0	0	0.044
	Q17	0.441	0.0735	0.441
	Q18	0	0.063	7.82
	Q19	0.22	0.104	0.104
	Q20	0.014	0.014	0.181
Relative Error		**2.1045**	**4.686**	**12.88**

Table 6.13 shows the relative cost error for three different considered cost models. Cm refers to the cost function described in Chapter 4 and Cmm is the simplified cost model from [55] and the last one is the PostgreSQL cost model. Err_{Cm}, Err_{Cmm} and $Err_{postgres}$

denote the relative error with the default cost model, the simplified and postgres cost models, respectively.

From Table 6.13, our cost model Cm selects the query plan (with the minimum query evaluation time in the plan space) in queries Q_1, Q_4, Q_8, Q_{10}, Q_{15}, Q_{16} and Q_{18} resulting in a cost error of 0. Similarly, the simplified cost model also selects the cheapest query plans in Q_6, Q_8, Q_{15} and Q_{16}. And PostgreSQL cost model select queries with the minimum query evaluation time in queries Q_4 and Q_5.

In all cases, Cm performs relatively better than Cmm and significantly better than postgres. PostgreSQL cost model in particular selects a very bad query plan for query 18.

To demonstrate the overall performance of each cost model, we define MRE (mean relative error) in Equation 5.11 which takes the average relative error.

$$\text{MRE} = \frac{1}{N} \sum_{i=1}^{N} \text{Err}_i$$

where N is the total number of queries.

Table 6.14 – MRE for cost model

Cost model	Mean Relative Error
Cm	0.105
Cmm	0.234
PostgreSQL	0.644

Results for MRE in Table 6.14 shows that the cost function described in this work is more accurate than the simplified cost model and PostgreSQL cost model.

6.4.11 Analyzing QEP: Observing Selective Operation Pushdown

One of the principles of query optimization is the optimal arrangement of operators of the query tree. This has been a subject of study for decades and it is categorized into the logical and physical optimizations. Logical optimization refers to process of generating optimal sequence of equivalent relational expressions or subexpressions for a query. Physical optimization is concerned with finding the best algorithm to implement for a given logical sequence of operators and the order in which the physical operations are performed.

In this chapter, we focused on logical optimization and we considered the operator ordering on the logical level in [26]. Some examples of these optimization include pushing filter operations down the query plan tree as possible and pushing filter through recursion and inside joins whenever possible.

For example, the query ?x A/B c involves joining path A and B and filtering with a constant c. The query can be translated into RA as follows;

1. σ_c (A $\bowtie$ B): join A and B, then filter for c

2. $\sigma_c(A) \bowtie \sigma_c(B)$: remove tuples from both relations A and B before joining them.

The second option is intuitively more efficient than the first translation since this option ensures that the number of tuples participating in the join is reduced. What this means in terms of performance is that the amount of memory space used in doing the unnecessary joining of extra tuples that will not satisfy condition c is avoided, leading to faster evaluation time.

For the query plans selected by our cost model, this arrangement and the ones described in [26] are most of the times retained.

In search for optimal plans, query optimizers cost models should favour this arrangement. However, in reality this is often not the case if poor or inaccurate statistics (e.g. cardinality) are used or if the cost function is not adequately designed. These deficiencies can lead to expensive query plans having cheap cost estimates and becoming a candidate query plan potentially selected by the optimizer.

In summary, we implemented the cost functions defined in Chapter 4 for recursive query evaluation and went through the different components of the cost model architecture and the integration of the improved cardinality estimation framework; SumRDF [36] into our estimation framework.

Using the validation framework presented in Chapter 5 as the guidelines for our practical experiments, the results of the evaluation of our cost estimation technique on various datasets and query forms has been given. We carried out several in-depth experiments, comparing our estimation technique with the state-of-the-art and demonstrated its effectiveness of our estimation techniques by showing that in most cases, our approach outperforms these systems by an order of magnitude.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we propose a cost estimation technique for recursive queries which uses data statistics. Estimating the maximum number of steps needed for a recursive evaluation to converge and return result is perhaps one of the most challenging aspect of cost estimation for the recursive operator (the fixpoint). In the first part of our contribution, rather than treating a recursive evaluation as a blackbox where recursion is considered to happen in a constant step, instead we present a step by step analysis of the operation in each step of the recursive evaluation. We use this knowledge of the iterative steps to then formulate a cost function for recursive query evaluation. We also extend the state-of-the-art cardinality estimation technique for non-recursive queries on RDF graphs for recursive queries cardinality estimation with the objective to improve the accuracy of the cardinality estimation and query plan quality.

Our second contribution is a cost validation framework where we propose a set of metrics, specifications and conditions for the optimality of query plan. These metrics are used to support the assessment of cost model query plan-selection quality. This framework does not only provide a way for assessing the effectiveness of a cost model but also a method for comparing different query optimizer's cost functions which is currently lacking. Rather than using just the query execution times for comparing the effectiveness of a cost model, we develop a technique that uses intervals of estimates and a ranking-based approach where cost models are ranked based on the quality of the selected plans.

Experiments with a prototype of the approach shows that this technique improves the performance of recursive query evaluation on popular relational database engines such as Postgres. This contribution is generic and can be implemented in any mainstream database management systems supporting recursive query evaluation. Finally, the work

presented here provides foundations for recursive query cost estimation.

7.2 Future Work

Selectivity and Cardinality Estimation

Selectivity estimation is important for the cardinality and cost model accuracy. In this thesis, we use the equi-width histogram for maintaining the selectivities of attributes. Histograms have a known limitation of not adequately capturing the correlation between (join) attributes from different tables and updating the histograms for these multi-tables attributes (using multi-dimension histogram) requires a lot of space and are often difficult to carry out. Thus, for future work, a sampling-based approach could be more efficient. We also plan to expand our experimental study to include more recent cardinality estimation models.

Cardinality Estimations for Specific Data Models

The cardinality estimation method considered in Chapter 4 is centered around recursive queries execution on relational model. In future work, it is also necessary to test this cardinality estimation together with the cost estimation techniques for recursive queries on data models such as knowledge or property graphs.

Graph Summarization

In this work, we made experiments with state-of-the-art RDF graph summarization technique for cardinality estimations and we have seen how accurate cardinality estimation can improve the quality of plan selection or at least ensure that selected plans are usually near the optimal ones. For large dataset, however, the summarization process is time-consuming. We plan to reduce the time required to generate summaries for a given dataset by moving the computation to a distributed systems like the Apache Spark. In the future, we would also like to experiment this technique in other query optimizers in order to improve their cost model performance and provide support for recursion. The idea being that, if cardinality estimation error is minimized, developers can focus more on formulating a better cost model and improving the accuracy of the cost estimation.

Cost Estimation and Plan Selection for the Distributed Setting

We would like to expand the scope of the cost model to allows the development of additional cost functions for join algorithms and scan. For example, the cost model presented in Chapter 4 only support sequential scan and natural join, a future work will include cost formulas for different types of join algorithms like hash join, sort-merge join, and broadcast and shuffle joins in the distributed setting. In order to improve effectiveness of query plan selection, another idea for the cost model would be considering the `topN`

cheapest (according to the cost model) query plan from the set of plans for a given query and then applying additional techniques to further determine a better plan among these. This could potentially eliminate bad plans from being selected for execution by the query optimizer. We also expect further work to be able to support distributed and cloud-based environments.

In Chapter 5, we present a cost validation framework that defines the metrics and specifications, estimation error models and the query plan ranking. We intend to implement these ideas in a benchmark that will be suitable for testing and validating different cost models. This will also help to establish the relationships between the accuracy of the cost models and the optimality of the selected query plan.

Learning-based Approach

Recently, we have seen many learning-based techniques that learn from query plan cost and query runtime information to make predictions for future query workloads. While these techniques have been applied in the context of non-recursive queries both on RDBMS and RDF data, it will be interesting to extend such techniques to recursive queries. The combination of learning techniques with traditional cost estimation can improve the quality of plan selection.

Bibliography

[1] E. F. Codd. "A relational model of data for large shared data banks." In: *Software pioneers*. Springer, 2002, pp. 263–294 (cited on page 2).

[2] Apache Software Foundation. *Hadoop*. Version 0.20.2. Feb. 19, 2010. URL: https://hadoop.apache.org (cited on page 2).

[3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. "Apache spark: a unified engine for big data processing." In: *Communications of the ACM* 59.11 (2016), pp. 56–65 (cited on page 2).

[4] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. "Spinning fast iterative data flows." In: *arXiv preprint arXiv:1208.0088* (2012) (cited on pages 2, 4, 20, 22, 60, 63).

[5] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. "A comparison of a graph database and a relational database: a data provenance perspective." In: *Proceedings of the 48th annual Southeast regional conference*. 2010, pp. 1–6 (cited on page 2).

[6] R. Angles and C. Gutierrez. "Survey of graph database models." In: *ACM Computing Surveys (CSUR)* 40.1 (2008), pp. 1–39 (cited on page 2).

[7] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access path selection in a relational database management system." In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34 (cited on pages 3, 6, 7, 17, 18, 25, 26, 29, 31, 32, 34, 35, 42, 44, 48, 53, 89, 90).

[8] E. Pitoura. "Query Optimization." In: *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, 2009, pp. 2272–2273. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_861. URL: https://doi.org/10.1007/978-0-387-39940-9_861 (cited on page 3).

[9] N. Bruno, C. Galindo-Legaria, and M. Joshi. "Polynomial heuristics for query optimization." In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE. 2010, pp. 589–600 (cited on page 3).

[10] S. Chaudhuri. "An overview of query optimization in relational systems." In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM. 1998, pp. 34–43 (cited on pages 3, 6, 29, 49, 53).

[11] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. "Heuristics-based query optimisation for SPARQL." In: *Proceedings of the 15th International Conference on Extending Database Technology*. 2012, pp. 324–335 (cited on pages 3, 18).

[12] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. "Cardinality estimation with local deep learning models." In: *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management*. 2019, pp. 1–8 (cited on pages 3, 28).

[13] Y. Park, S. Zhong, and B. Mozafari. "Quicksel: Quick selectivity learning with mixture models." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1017–1033 (cited on pages 3, 28).

[14] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. "Learned cardinalities: Estimating correlated joins with deep learning." In: *arXiv preprint arXiv:1809.00677* (2018) (cited on pages 3, 28).

[15] F. Li, B. Wu, K. Yi, and Z. Zhao. "Wander join: Online aggregation via random walks." In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 615–629 (cited on pages 3, 28).

[16] T. Neumann and G. Moerkotte. "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins." In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 984–994 (cited on pages 3, 16, 28).

[17] F. Cacace, S. Ceri, and M. A. Houtsma. "An overview of parallel strategies for transitive closure on algebraic machines." In: *Workshop on Parallel Database Systems*. Springer. 1990, pp. 44–62 (cited on pages 4, 20, 63).

[18] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. "Sqlgraph: An efficient relational-based property graph store." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 1887–1901 (cited on pages 4, 16).

[19] S. Floratos, Y. Zhang, Y. Yuan, R. Lee, and X. Zhang. "SQLoop: High Performance Iterative Processing in Data Management." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 1039–1051 (cited on page 4).

[20] M. Lawal, P. Genevès, and N. Layaïda. "A Cost Estimation Technique for Recursive Relational Algebra." In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2020, pp. 3297–3300 (cited on pages 4, 6, 79, 96).

[21] Y. E. Ioannidis. "On the computation of the transitive closure of relational operators." In: (cited on pages 4, 7, 20, 37, 42–44, 63).

[22] U. Güntzer, W. Kießling, and R. Bayer. "On the evaluation of recursion in (deductive) database systems by efficient differential fixpoint iteration." In: *1987 IEEE Third International Conference on Data Engineering*. IEEE. 1987, pp. 120–129 (cited on pages 4, 20, 22, 60).

[23] R. S. Lanzelotte, P. Valduriez, and M. Zaït. "Optimization of object-oriented recursive queries using cost-controlled strategies." In: *ACM SIGMOD Record*. Vol. 21. 2. ACM. 1992, pp. 256–265 (cited on pages 4, 6, 20, 37, 42–44).

[24] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. "Map-reduce extensions and recursive queries." In: *Proceedings of the 14th international conference on extending database technology*. 2011, pp. 1–8 (cited on pages 4, 20, 63).

[25] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. "Big data analytics with datalog queries on spark." In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1135–1149 (cited on pages 4, 20).

[26] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda. "On the Optimization of Recursive Relational Queries: Application to Graph Queries." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 681–697 (cited on pages 4, 5, 8, 9, 11, 16, 20, 22, 23, 43, 53, 54, 60, 70, 87, 90, 91, 94, 95, 103, 113, 114).

[27] V.-Q. Nguyen and K. Kim. "Estimating the evaluation cost of regular path queries on large graphs." In: *Proceedings of the Eighth International Symposium on Information and Communication Technology*. 2017, pp. 92–99 (cited on pages 4, 16, 17).

[28] S. R. Mihaylov, Z. G. Ives, and S. Guha. "REX: recursive, delta-based data-centric computation." In: *arXiv preprint arXiv:1208.0089* (2012) (cited on page 4).

[29] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. "The HaLoop approach to large-scale iterative data analysis." In: *The VLDB Journal* 21.2 (2012), pp. 169–190 (cited on page 4).

[30] P. J. Haas, J. F. Naughton, S Seshadri, and A. N. Swami. "Selectivity and cost estimation for joins based on random sampling." In: *Journal of Computer and System Sciences* 52.3 (1996), pp. 550–569 (cited on pages 6, 7, 28, 53).

[31] M. Golfarelli and L. Baldacci. "A cost model for spark sql." In: *IEEE Transactions on Knowledge and Data Engineering* (2018) (cited on pages 6, 27, 32, 36, 44).

[32] 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: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 1081–1092 (cited on pages 6, 25, 26, 28, 29, 35, 38, 44, 46, 49, 53, 74).

[33] *Apache Flink*. https://github.com/apache/flink/blob/master/flink-optimizer/src/main/java/org/apache/flink/optimizer/costs/DefaultCostEstimator.java (cited on page 6).

[34] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 221–230 (cited on page 6).

[35] A. Swami and K. B. Schiefer. "On the estimation of join result sizes." In: *International Conference on Extending Database Technology*. Springer. 1994, pp. 287–300 (cited on pages 7, 26, 58, 69).

[36] G. Stefanoni, B. Motik, and E. V. Kostylev. "Estimating the cardinality of conjunctive queries over RDF data using graph summarisation." In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 1043–1052 (cited on pages 7–9, 28, 46, 49, 66–68, 81, 90, 91, 114).

[37] M. Freitag and T. Neumann. "Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates." In: *ratio* 1 (2019), pp. 1–39 (cited on page 7).

[38] I. Trummer. "Exact cardinality query optimization with bounded execution cost." In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 2–17 (cited on pages 7, 28, 49).

[39] F. Wolf, N. May, P. R. Willems, and K.-U. Sattler. "On the calculation of optimality ranges for relational query execution plans." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 663–675 (cited on pages 7, 28).

[40] S. Heule, M. Nunkesser, and A. Hall. "HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm." In: *Proceedings of the 16th International Conference on Extending Database Technology*. ACM. 2013, pp. 683–692 (cited on pages 7, 49).

[41] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. "Exact cardinality query optimization for optimizer testing." In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 994–1005 (cited on pages 7, 28, 49).

[42] N. Schweikardt, T. Schwentick, and L. Segoufin. "Database theory: Query languages." In: () (cited on page 13).

[43] D. C. Faye, O. Curé, and G. Blin. "A survey of RDF storage approaches." In: (2012) (cited on page 15).

[44] B. McBride. "The resource description framework (RDF) and its vocabulary description language RDFS." In: *Handbook on ontologies*. Springer, 2004, pp. 51–65 (cited on page 16).

[45] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. "Expressive languages for path queries over graph-structured data." In: *ACM Transactions on Database Systems (TODS)* 37.4 (2012), pp. 1–46 (cited on pages 16, 17).

[46] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. "A graphical query language supporting recursion." In: *ACM SIGMOD Record* 16.3 (1987), pp. 323–330 (cited on pages 16, 17).

[47] P. Barceló, D. Figueira, and L. Libkin. "Graph logics with rational relations and the generalized intersection problem." In: *2012 27th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2012, pp. 115–124 (cited on pages 16, 17).

[48] W. Martens and T. Trautner. "Evaluation and enumeration problems for regular path queries." In: *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018 (cited on page 17).

[49] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. "Extensible query processing in Starburst." In: *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 1989, pp. 377–388 (cited on page 17).

[50] G. Graefe and W. J. McKenna. "The volcano optimizer generator: Extensibility and efficient search." In: *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE. 1993, pp. 209–218 (cited on pages 17, 18, 90).

[51] G. Graefe. "The cascades framework for query optimization." In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29 (cited on pages 17, 18, 90).

[52] C. Ordonez. "Optimization of linear recursive queries in SQL." In: *IEEE Transactions on Knowledge and Data Engineering* 22.2 (2009), pp. 264–277 (cited on page 19).

[53] A. V. Aho and J. D. Ullman. "Universality of data retrieval languages." In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1979, pp. 110–119 (cited on page 22).

[54] F. Cacace, S. Ceri, S. Crespi-Reghizzi, G. Gottlob, G. Lamperti, L. Lavazza, L. Tanca, and R. V. Zicari. "ALGRES: An Extended Relational Database System for the Specification and Prototyping of Complex Applications." In: *CA (i) SE*. 1989 (cited on pages 22, 60).

[55] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215 (cited on pages 25, 26, 28, 35, 39, 46, 49, 53, 68, 73, 74, 81, 104, 110, 112).

[56] O. Ivanov and S. Bartunov. "Adaptive cardinality estimation." In: *arXiv preprint arXiv:1711.08330* (2017) (cited on pages 25, 49).

[57] Y. Qin, K. Salem, and A. K. Goel. "Towards adaptive costing of database access methods." In: *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE. 2007, pp. 469–477 (cited on page 26).

[58] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. "Learning-based query performance modeling and prediction." In: *2012 IEEE 28th International Conference on Data Engineering*. IEEE. 2012, pp. 390–401 (cited on pages 26, 27, 38, 44, 82).

[59] H. Herodotou. "Hadoop performance models." In: *arXiv preprint arXiv:1106.0940* (2011) (cited on pages 26, 27, 32, 33, 38, 44).

[60] 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: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 592–603 (cited on pages 26, 37, 44).

[61] L. F. Mackert. "R* optimizer validation and performance evaluation for distributed queries." In: *Readings in database systems* (1998), pp. 351–361 (cited on pages 26, 32, 34, 38, 42, 44, 53).

[62] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. "Spark sql: Relational data processing in spark." In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM. 2015, pp. 1383–1394 (cited on pages 27, 35, 42, 44).

[63] J. Sun and G. Li. "An end-to-end learning-based cost estimator." In: *arXiv preprint arXiv:1906.02560* (2019) (cited on pages 27, 36, 44).

[64] M. Zneika. "Querying semantic web/linked data graphs using summarization." Doctoral dissertation. Université de Cergy Pontoise, 2019 (cited on page 28).

[65] M. P. Consens, V. Fionda, S. Khatchadourian, and G. Pirro. "S+ epps: construct and explore bisimulation summaries, plus optimize navigational queries; all on existing sparql systems." In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 2028–2031 (cited on page 28).

[66] S. Campinas, T. E. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. "Introducing RDF graph summary with application to assisted SPARQL formulation." In: *2012 23rd International Workshop on Database and Expert Systems Applications*. IEEE. 2012, pp. 261–266 (cited on page 28).

[67] M. Konrath, T. Gottron, S. Staab, and A. Scherp. "Schemex—efficient construction of a data catalogue by stream-based indexing of linked data." In: *Journal of Web Semantics* 16 (2012), pp. 52–58 (cited on page 28).

[68] F. Picalausa, Y. Luo, G. H. Fletcher, J. Hidders, and S. Vansummeren. "A structural approach to indexing triples." In: *Extended Semantic Web Conference*. Springer. 2012, pp. 406–421 (cited on page 28).

[69] G. Troullinou, H. Kondylakis, E. Daskalaki, and D. Plexousakis. "RDF digest: Efficient summarization of RDF/S KBs." In: *European Semantic Web Conference*. Springer. 2015, pp. 119–134 (cited on page 28).

[70] E. Motta, P. Mulholland, S. Peroni, M. d'Aquin, J. M. Gomez-Perez, V. Mendez, and F. Zablith. "A novel approach to visualizing and navigating ontologies." In: *International Semantic Web Conference*. Springer. 2011, pp. 470–486 (cited on page 28).

[71] T. Tran, G. Ladwig, and S. Rudolph. "Managing structured and semistructured RDF data using structure indexes." In: *IEEE Transactions on Knowledge and Data Engineering* 25.9 (2012), pp. 2076–2089 (cited on page 28).

[72] A. Maduko, K. Anyanwu, A. Sheth, and P. Schliekelman. "Estimating the cardinality of RDF graph patterns." In: *Proceedings of the 16th international conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: Association for Computing Machinery, May 2007, pp. 1233–1234. ISBN: 9781595936547. DOI: 10.1145/1242572.1242782. URL: https://doi.org/10.1145/1242572.1242782 (visited on 01/26/2021) (cited on page 28).

[73] X. Chen and J. C. Lui. "Mining graphlet counts in online social networks." In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12.4 (2018), pp. 1–38 (cited on page 28).

[74] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. "Towards a learning optimizer for shared clouds." In: *Proceedings of the VLDB Endowment* 12.3 (2018), pp. 210–222 (cited on page 28).

[75] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. "An empirical analysis of deep learning for cardinality estimation." In: *arXiv preprint arXiv:1905.06425* (2019) (cited on page 28).

[76] W. Cai, M. Balazinska, and D. Suciu. "Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities." In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 18–35 (cited on page 28).

[77] F. Rusu and A. Dobra. "Sketches for size of join estimation." In: *ACM Transactions on Database Systems (TODS)* 33.3 (2008), pp. 1–46 (cited on page 28).

[78] V. Poosala, V. Ganti, and Y. E. Ioannidis. "Approximate query answering using histograms." In: *IEEE Data Eng. Bull.* 22.4 (1999), pp. 5–14 (cited on page 28).

[79] Y. E. Ioannidis and V. Poosala. "Balancing histogram optimality and practicality for query result size estimation." In: *Acm Sigmod Record* 24.2 (1995), pp. 233–244 (cited on page 28).

[80] J. Acharya, I. Diakonikolas, C. Hegde, J. Z. Li, and L. Schmidt. "Fast and near-optimal algorithms for approximating distributions by histograms." In: *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2015, pp. 249–263 (cited on page 28).

[81] C. L. Canonne. "Are few bins enough: Testing histogram distributions." In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 455–463 (cited on page 28).

[82] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. "Improved histograms for selectivity estimation of range predicates." In: *ACM Sigmod Record* 25.2 (1996), pp. 294–305 (cited on page 28).

[83] R. Kaushik and D. Suciu. "Consistent histograms in the presence of distinct value counts." In: *Proceedings of the VLDB Endowment* 2.1 (2009), 850–861. DOI: 10.14778/1687627.1687723 (cited on page 28).

[84] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen. "Join size estimation subject to filter conditions." In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1530–1541 (cited on page 28).

[85] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. "Random sampling over joins revisited." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1525–1539 (cited on page 28).

[86] P. J. Haas, J. F. Naughton, S Seshadri, and A. N. Swami. "Fixed-precision estimation of join selectivity." In: *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1993, pp. 190–201 (cited on page 28).

[87] S. Yin, A. Hameurlain, and F. Morvan. "Robust query optimization methods with respect to estimation errors: A survey." In: *ACM Sigmod Record* 44.3 (2015), pp. 25–36 (cited on pages 28, 73).

[88] S. Manegold et al. *Understanding, modeling, and improving main-memory database performance*. Universiteit van Amsterdam [Host], 2002 (cited on pages 29, 31).

[89] S. Sakr and G. Al-Naymat. "Relational processing of RDF queries: a survey." In: *ACM SIGMOD Record* 38.4 (2010), pp. 23–28 (cited on page 32).

[90] K. V. Emani and S Sudarshan. "Cobra: A framework for cost-based rewriting of database applications." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 689–700 (cited on pages 36, 44).

[91] L. N. P. Obermeier and L. Nixon. "A cost model for querying distributed rdf-repositories with sparql." In: *Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense Tenerife, Spain*. 2008 (cited on pages 36, 44).

[92] M. Saleem, A. Potocki, T. Soru, O. Hartig, and A.-C. N. Ngomo. "CostFed: Cost-based query optimization for SPARQL endpoint federation." In: *Procedia Computer Science* 137 (2018), pp. 163–174 (cited on pages 36, 44).

[93] F. Wolf, M. Brendle, N. May, P. R. Willems, K.-U. Sattler, and M. Grossniklaus. "Robustness metrics for relational query execution plans." In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1360–1372 (cited on pages 36, 73, 76).

[94] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. "Statistical learning techniques for costing XML queries." In: *Proceedings of the 31st international conference on Very large data bases*. 2005, pp. 289–300 (cited on page 38).

[95] R. Hasan and F. Gandon. "A machine learning approach to sparql query performance prediction." In: *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. Vol. 1. IEEE. 2014, pp. 266–273 (cited on page 38).

[96] W. Wu. "A Note On Operator-Level Query Execution Cost Modeling." In: *arXiv preprint arXiv:2003.04410* (2020) (cited on page 41).

[97] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S Sudarshan. "Cost-based optimization for magic: Algebra and implementation." In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 1996, pp. 435–446 (cited on page 44).

[98] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W.-S. Han. "G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1099–1114 (cited on page 49).

[99] A. Ouared, Y. Ouhammou, and L. Bellatreche. "Costdl: a cost models description language for performance metrics in database." In: *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE. 2016, pp. 187–190 (cited on pages 51, 85).

[100] Š. Čebirić, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. "Summarizing semantic graphs: a survey." In: *The VLDB Journal* 28.3 (2019), pp. 295–327 (cited on page 66).

[101] A. Ouared, Y. Ouhammou, and L. Bellatreche. "MetricStore repository: on the leveraging of performance metrics in databases." In: *Proceedings of the Symposium on Applied Computing*. 2017, pp. 1820–1825 (cited on page 73).

[102] G. Moerkotte, T. Neumann, and G. Steidl. "Preventing bad plans by bounding the impact of cardinality estimation errors." In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 982–993 (cited on pages 80, 98, 100).

[103] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. "Querying graphs." In: *Synthesis Lectures on Data Management* 10.3 (2018), pp. 1–184 (cited on page 95).

[104] M. Stonebraker and L. A. Rowe. "The design of POSTGRES." In: *ACM Sigmod Record* 15.2 (1986), pp. 340–355 (cited on page 95).

[105] F. M. Suchanek, G. Kasneci, and G. Weikum. "Yago: A Core of Semantic Knowledge." In: *16th International Conference on the World Wide Web*. 2007, pp. 697–706 (cited on pages 96, 103).

[106] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. "gMark: Schema-Driven Generation of Graphs and Queries." In: *IEEE Transactions on Knowledge and Data Engineering* 29.4 (2017), pp. 856–869 (cited on pages 96, 110, 111).

APPENDIX 1 QUERIES

A.1 Uniprot Queries

Q1 ?x0, ?x3, ?x1, ?x4, ?x2 <- ?x0 ((-HasKeyword/-Interacts)/Reference)|((-HasKeyword/Interacts)/Reference) ?x1, ?x1 (((-Reference/Interacts)/Interacts)/Interacts)|((-Reference/-Interacts)/Interacts) ?x2, ?x2 (((-Interacts/Interacts)/Interacts)/Interacts)+ ?x3, ?x3 (((((-Interacts/Interacts)/-Interacts)/Interacts)|(((Interacts/Interacts)/-Interacts)/-Interacts))|((Interacts/Interacts)/Interacts))+ ?x4

Q2 ?x0, ?x2, ?x1 <- ?x0 (((((Interacts/Interacts)/Interacts)/-Interacts)|(-Interacts/Interacts))|((Interacts/-Interacts)/-Interacts))+ ?x1, ?x0 ((((-Interacts/Interacts)/-Interacts)/Interacts)|(-Interacts/Interacts))+ ?x2, ?x0 (((((Interacts/Interacts)/Interacts)/-Interacts)|((Interacts/-Interacts)/-Interacts))|(Interacts/Interacts))+ ?x3

Q3 ?x0, ?x4 <- ?x0 (((-Interacts/-Interacts)/Interacts)|(((-Interacts/-Interacts)/-Interacts)/Interacts))+ ?x1, ?x1 ((-Interacts/-Interacts)/Interacts)+ ?x2, ?x2 (Interacts/Reference)|((Interacts/Interacts)/Reference) ?x3, ?x3 (AuthoredBy|AuthoredBy)|AuthoredBy ?x4

Q4 ?x0, ?x2, ?x1 <- ?x0 (((-Interacts/-Interacts)/Interacts)/Interacts)+ ?x1, ?x1 (((-Interacts/-Interacts)/-Interacts)/-Interacts)+ ?x2, ?x0 (Interacts/-Interacts)+ ?x3, ?x3 ((Reference/-Reference)/Interacts)/-Interacts ?x2

Q5 ?x0, ?x3 <- ?x0 (-AuthoredBy/-Reference)/OccursIn ?x1, ?x1 ((-OccursIn/OccursIn)/-OccursIn)/-Interacts ?x2, ?x2 (((Interacts/Interacts)|((-Interacts/Interacts)/Interacts))|((-Interacts/-Interacts)/-Interacts))+ ?x3

Q6 ?x0, ?x2, ?x1 <- ?x0 ((((-HasKeyword/HasKeyword)/-HasKeyword)/Interacts)|(-HasKeyword/Interacts))|(((-HasKeyword/OccursIn)/-OccursIn)/-Interacts) ?x1, ?x1 (((-Interacts/-Interacts)/-Interacts)|(((-Interacts/-Interacts)/-Interacts)/-Interacts))+ ?x2, ?x2 (-Interacts/Interacts)+ ?x3

Q7 ?x0, ?x3 <- ?x0 ((((Interacts/Interacts)/Interacts)/-Interacts)|((-Interacts/Interacts)/Interacts))+ ?x1, ?x1 (-Interacts/-Interacts)|(((Interacts/-Interacts)/-Interacts)/-Interacts) ?x2, ?x2 (((-Interacts/Interacts)/-Interacts)/OccursIn)|(((-Interacts/-Interacts)/-Interacts)/OccursIn) ?x3

Q8 ?x0 <- ?x0 (((-OccursIn/-Interacts)/Reference)|((-OccursIn/-Interacts)/Reference))|((-OccursIn/Interacts)/Reference) ?x1, ?x1 ((((PublishedIn/-PublishedIn)/-Reference)/-Interacts)|((-Reference/OccursIn)/-OccursIn))|((AuthoredBy/-AuthoredBy)/-Reference) ?x2, ?x2 ((((Interacts/Interacts)/Interacts)/Interacts)|((Interacts/Interacts)/Interacts))+ ?x3, ?x3 (-Interacts/-Interacts)+ ?x4

Q9 ?x3, ?x1, ?x2, ?x0 <- ?x0 (((Interacts/Interacts)/-Interacts)|((-Interacts/-Interacts)/-Interacts))+ ?x1, ?x1 ((((-Interacts/Interacts)/-Interacts)|(-Interacts/Interacts))|(((Interacts/Interacts)/-Interacts)/Interacts))+ ?x2, ?x0 (((Interacts/Interacts)|(((Interacts/Interacts)/Interacts)/-Interacts))|(((-Interacts/Interacts)/-Interacts)/Interacts))+ ?x3, ?x3 (((((-Interacts/-Interacts)/Interacts)/-Interacts)|((-Interacts/-Interacts)/-Interacts))|((Interacts/Interacts)/-Interacts))+ ?x2

Q10 ?x0, ?x2, ?x1 <- ?x0 (Interacts/Interacts)/EncodedOn ?x1, ?x0 (((-Interacts/Interacts)|(-Interacts/-Interacts))|((-Interacts/Interacts)/Interacts))+ ?x2, ?x2 ((-Interacts/Reference)/-Reference)/HasKeyword ?x1

A.2 Shop Queries

Q1 ?x0, ?x1, ?x2, ?x3 <- ?x0 (((employee/follows)/-artist)|(((offers/-offers)/contactPoint)/-artist))|((employee/follows)/-artist) ?x1, ?x1 ((-purchaseFor/purchaseFor)|(((homepage/-homepage)/-reviewer)/-hasReview))+ ?x2, ?x2 (homepage/-homepage)/homepage ?x3

Q2 ?x0, ?x2 <- ?x0 ((-nationality/like)/hasGenre)|((-editor/includes)/hasGenre) ?x1, ?x1 (((-hasGenre/conductor)/age)|((-hasGenre/author)/age))|((-hasGenre/-like)/age) ?x2, ?x0 -nationality/-reviewer ?x3, ?x3 (((-hasReview/hasReview)|(((-hasReview/-purchaseFor)/purchaseFor)/hasReview))|(((-hasReview/editor)/-author)/hasReview))+ ?x2

Q3 ?x0 <- ?x0 ((((homepage/-subscribes)/-editor)|(((-purchaseFor/purchaseFor)/-includes)/includes))|((homepage/-subscribes)/-editor))+ ?x1, ?x1

((editor/friendOf)/-author)+ ?x2, ?x2 (((author/makesPurchase)/purchaseFor)|(((-includes/includes)/conductor)/-author))+ ?x3, ?x3 ((((keywords/-performer)/contentRating)/-hits)|(printSection/-hits))|((text/-composer)/homepage) ?x4

Q4 ?x0, ?x2 <- ?x0 ((-director/director)|((-actor/director)/friendOf))+ ?x1, ?x1 (-friendOf/-friendOf)+ ?x2, ?x0 (((telephone/-eligibleQuantity)|((like/release)/-validForm))|((givenName/-title)/-includes))+ ?x3, ?x2 ((like/numberOfPages)/-eligibleQuantity)+ ?x4

Q5 ?x0, ?x1 <- ?x0 ((-nationality/-director)/language)|(((-parentContry/-location)/-director)/language) ?x1, ?x0 ((((-nationality/userId)/-isbn)/homepage)|((-nationality/like)/trailer))|(((-nationality/-friendOf)/-friendOf)/homepage) ?x2, ?x2 ((-homepage/author)/homepage)+ ?x1

Q6 ?x0, ?x3, ?x1, ?x4, ?x2 <- ?x0 (((((purchaseFor/director)/-actor)/-purchaseFor)|(purchaseFor/-purchaseFor))|((-makesPurchase/-editor)/-purchaseFor))+ ?x1, ?x1 (((purchaseFor/-includes)/includes)/-like)|(purchaseFor/-like) ?x2, ?x2 ((friendOf/friendOf)/follows)/-artist ?x3, ?x3 ((contentRating/-numberOfPages)/title)|(-like/jobTitle) ?x4

Q7 ?x1, ?x2, ?x0 <- ?x0 ((-contentRating/-includes)|((-contentRating/recordNumber)/-eligibleQuantity))|(-opus/-includes) ?x1, ?x1 (((includes/hasReview)/-hasReview)/-includes)+ ?x2, ?x0 ((((-printColumn/description)/-description)/text)|(((-contentRating/duration)/-contentRating)/keywords))|(-openingHours/name) ?x3, ?x3 (((-keywords/-includes)/editor)|(-jobTitle/nationality))|((-description/artist)/nationality) ?x2

Q8 ?x2, ?x0, ?x1 <- ?x0 ((purchaseFor/-purchaseFor)|((purchaseFor/director)/makesPurchase))+ ?x1, ?x0 (((price/-printEdition)|((purchaseFor/title)/-caption))|(price/-wordCount))+ ?x2, ?x1 ((price/-contentSize)|((purchaseFor/expires)/-datePublished))+ ?x3

Q9 ?x2, ?x0, ?x1 <- ?x0 ((purchaseFor/contentRating)/-contentSize)/hasGenre ?x1, ?x1 ((-hasGenre/producer)/-description)/-includes ?x2, ?x2 (includes/-includes)+ ?x3, ?x3 ((((includes/-purchaseFor)/purchaseFor)/-includes)|(includes/-includes))+ ?x4

Q10 ?x0, ?x3 <- ?x0 (((-expires/composer)/-title)|(-validForm/includes))|((-expires/homepage)/-homepage) ?x1, ?x1 (-includes/includes)+ ?x2, ?x2 ((((caption/-caption)/hasGenre)/type)|(((title/-text)/hasGenre)/type))|(hasGenre/type) ?x3

A.3 Yago Queries

Q1 ?x <- ?x <isMarriedTo>/<livesIn>/<isLocatedIn>+/<dealsWith>+ <Argentina>

Q2 ?x <- ?x <hasChild>/<livesIn>/<isLocatedIn>+/<dealsWith>+ <Japan>

Q3 ?x <- ?x <influences>/<livesIn>/<isLocatedIn>+/<dealsWith>+ <Sweden>

Q4 ?x <- ?x <livesIn>/<isLocatedIn>+/<dealsWith>+ <United_States>

Q5 ?x <- ?x <hasSuccessor>/<livesIn>/<isLocatedIn>+/<dealsWith>+ <India>

Q6 ?x <- ?x <hasPredecessor>/<livesIn>/<isLocatedIn>+/<dealsWith>+ <Germany>

Q7 ?x <- ?x <hasAcademicAdvisor>/<livesIn>/<isLocatedIn>+/<dealsWith>+ <Netherlands>

Q8 ?x <- ?x <isLocatedIn>+/<dealsWith>+ <United_States>

Q9 ?x <- ?x (<actedIn>/-<actedIn>)+ <Kevin_Bacon>

Q10 ?area <- <wikicategory_Capitals_in_Europe> -rdf:type/(<isLocatedIn>+/<dealsWith>|<dealsWith>) ?area

Q11 ?a, ?b <- ?a <isLocatedIn>+/<dealsWith> ?b

Q12 ?a, ?b <- ?a <isLocatedIn>+/<dealsWith>+ ?b

Q13 ?a, ?b, ?c <- ?a <wasBornIn>/<isLocatedIn>+ ?b, ?b <isConnectedTo>+ ?c

Q14 ?a, ?b, ?c <- ?a (<isLocatedIn>|<isConnectedTo>)+ ?b, ?c <wasBornIn> ?a

Q15 ?a, ?b, ?c <- ?a <isLocatedIn>+ ?b, ?b <isConnectedTo>+ ?c, ?a <wasBornIn> ?c

Q16 ?a, ?c <- ?a <wasBornIn>/<isLocatedIn>+ <Japan>, ?a rdf:type/rdfs:subClassOf ?c

Q17 ?a <- ?a <isLocatedIn>+/(<isConnectedTo>|<dealsWith>)+ <Japan>

Q18 ?a, ?c <- ?a <isLocatedIn>+ <Japan>, ?a <isConnectedTo>+ ?c

Q19 ?a <- ?a <isLocatedIn>+/<isLocatedIn> <Japan>

Q20 ?a <- ?a <isLocatedIn>+/<isConnectedTo>+/<dealsWith>+ <Japan>

Appendix B

Appendix 2 Word Cloud for the Thesis

Figure B.1 – Word cloud for Chapter 1 — Conclusion