NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Query Optimization on Distributed Databases

Eleftherios P. Katiforis

Supervisors:

Panagiotis Stamatopoulos, Assistant Professor, NKUA
Stasinos Konstantopoulos, Associate Researcher, NCSR «Demokritos»

ATHENS

OCTOBER 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση Ερωτημάτων σε Κατανεμημένες Βάσεις Δεδομένων

Ελευθέριος Π. Κατηφόρης

Επιβλέποντες:

Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής , ΕΚΠΑ
Στασινός Κωνσταντόπουλος, Συνεργαζόμενος Ερευνητής, ΕΚΕΦΕ «Δημόκριτος»

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2018

BSc THESIS

Query Optimization on Distributed Databases

Eleftherios P. Katiforis
R.N.: 1115201300065

Supervisors:

Panagiotis Stamatopoulos, Assistant Professor, NKUA
Stasinos Konstantopoulos, Associate Researcher, NCSR «Demokritos»

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση Ερωτημάτων σε Κατανεμημένες Βάσεις Δεδομένων

Ελευθέριος Π. Κατηφόρης
Α.Μ.: 1115201300065

Επιβλέποντες:

Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής , ΕΚΠΑ
Στασινός Κωνσταντόπουλος, Συνεργαζόμενος Ερευνητής, ΕΚΕΦΕ «Δημόκριτος»

ABSTRACT

In recent years the Web has evolved from a global information space of linked documents to a web of linked data. The number of data sources and the amount of data published has been exploding, covering diverse domains such as people, companies, publications, popular culture and online communities, life sciences, governmental and statistical data, and many more. So nowadays it is heavily required to apply optimization techniques on the systems querying these data. Efficient query processing depends on the construction of an efficient query plan to guide query execution. Detailed instance-level metadata about the data sources and statistics among the data distribution are used to estimate the cost of different query plans and select the optimal one. Query optimizers in query processing systems typically rely on histograms, data structures that approximate data distribution, in order to be able to apply their cost model. We noticed that there were cases where optimizer had really bad performance caused by the bad estimations of the histogram. These cases are rare,usually a big outlier is involved, and this is the reason why adaptive histograms can not deal with them. Therefore in this thesis we detected such cases and created a method to make histogram provide the optimizer better statistics on these rare edge cases. Even though this had a negative impact to the average case the improvement on the edge cases was more significant.

SUBJECT AREA: Database Query Optimization

KEYWORDS: histograms, machine learning, dynamic programming, statistics, semantic web, open data

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια, το Διαδίκτυο έχει εξελιχθεί από ένα παγκόσμιο χώρο πληροφοριών αποτελούμενο από συνδεδεμένα έγγραφα σε έναν παγκόσμιο ιστό συνδεδεμένων δεδομένων. Ο αριθμός των πηγών δεδομένων και ο όγκος των δημοσιευμένων δεδομένων έχει εκραγεί, καλύπτοντας διάφορους τομείς όπως ανθρώπους, εταιρείες, δημοσιεύσεις, λαϊκή κουλτούρα και διαδικτυακές κοινότητες, επιστήμες ζωής, κυβερνητικά και στατιστικά στοιχεία και πολλά άλλα. Συνεπώς, σήμερα απαιτείται έντονα η εφαρμογή τεχνικών βελτιστοποίησης στα συστήματα που διερευνούν τα δεδομένα αυτά. Η αποτελεσματική επεξεργασία ενός ερωτήματος εξαρτάται από την κατασκευή ενός αποτελεσματικού πλάνου για την εκτέλεση του ερωτήματος. Λεπτομερή μεταδεδομένα σχετικά με τις πηγές δεδομένων και τα στατιστικά στοιχεία σχετικά με την κατανομή των δεδομένων χρησιμοποιούνται για την εκτίμηση του κόστους διαφορετικών πλάνων εκτέλεσης ερωτημάτων και επιλογή του βέλτιστου. Οι βελτιστοποιητές ερωτημάτων στα συστήματα επεξεργασίας ερωτημάτων συνήθως βασίζονται σε ιστογράμματα, δομές δεδομένων που απεικονίζουν τη κατανομή των δεδομένων, προκειμένου να μπορέσουν να εφαρμοστούν το μοντέλο υπολογισμού του κόστους των διαφορετικών πλάνων. Παρατηρήσαμε ότι υπήρξαν περιπτώσεις όπου ο βελτιστοποιητής είχε πραγματικά κακή απόδοση που προκλήθηκε από τις κακές εκτιμήσεις του ιστογράμματος. Αυτές οι περιπτώσεις είναι σπάνιες, συνήθως οφείλονται σε μια ακραία τιμή, και αυτός είναι ο λόγος για τον οποίο τα προσαρμοστικά ιστογράμματα δεν μπορούν να τις αντιμετωπίσουν. Επομένως, σε αυτή την πτυχιακή ανιχνεύσαμε τέτοιες περιπτώσεις και δημιουργήσαμε μια μέθοδο για την βελτίωση των εκτιμήσεων του ιστογράμματος σε τέτοιες σπάνιες περιπτώσεις. Παρόλο που αυτό είχε αρνητικό αντίκτυπο στη μέση περίπτωση, η βελτίωση στις ακραίες περιπτώσεις ήταν πιο σημαντική.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Βελτιστοποίηση Ερωτημάτων Βάσεων Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ιστογράμματα, μηχανική μάθηση, δυναμικός προγραμματισμός, στατιστική, σημασιολογικός ιστός, ανοικτά δεδομένα

ACKNOWLEDGEMENTS

I would like to thank all supervisors who work together for this thesis. Specifically, they are Prof. Panagiotis Stamatopoulos, Dr. Stasinos Konstantopoulos and Dr. Angelos Charalambidis. All of them, are individuals with huge scientific experience on the field that they research. They are committed and want the work to be done right, that makes them professionals. This characteristic makes them role models for me. Above all, they have the intention to share their knowledge with others, helping them to improve. Through procedure of this thesis, I learned to approach and research better, unknown objects and to reveal new. I'm grateful for guidance, immense support and excellent collaboration that we had.

Athens, OCTOBER 2018
Eleftherios Katiforis

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis took place in Athens of Greece between NOVEMBER 2017 and OCTOBER 2018. This work is consisted by four parts, Research part, Implementation part, Experiments part and Writing part. During the first part, I had to familiarize with related work and terminology on field of database query optimization and histogram construction and maintenance and come up with an idea to improve the existing techniques. At the implementation phase it was needed to extend the existing software with the implementation of our idea. The experiments were done using this software and data provided from DBpedia datasets. All of the code is written in Java language. Finally, this text was written in order to present current work. It's described detailed both theoretical background and experiments that are executed. So, the reader can be navigated smoothly in all procedure of this thesis.

1. INTRODUCTION

In recent years the Web has evolved from a global information space of linked documents to a web of linked data. The number of data sources and the amount of data published has been exploding, covering diverse domains such as people, companies, publications, popular culture and online communities, the life sciences genes, governmental and statistical data, and many more.

So nowadays it is heavily required to apply optimization techniques on the systems querying these data. But how can this be done when speaking for such amounts of data?

Declarative query languages allow to easily express complex queries without knowing about the details of the physical data organization of the database. Advanced query processing technology transforms the high-level queries into efficient lower-level query execution strategies. The query transformation should achieve both correctness and efficiency. The main difficulty is to achieve efficiency and this is also one of the most important tasks of any database management system.

Efficient query processing depends on the construction of an efficient query plan to guide query execution. When a query is given in a declarative language a query processing system can select from a variety of many execution plans ,especially when speaking for a complex query, to retrieve the answer. Of course all these plans will retrieve the same answer but their cost may differ in orders of magnitude either in computational resources or execution time. Detailed instance-level metadata about the data sources and statistics among the data distribution are used to estimate the cost of different query plans and select the optimal one. Query optimizers in query processing systems typically rely on histograms, data structures that approximate data distribution, in order to be able to apply their cost model.

There has been a lot of work on adaptive query processing systems that update their histograms by observing and analyzing the results of the queries that constitute the client-requested workload, as opposed to maintenance workload only for updating the histograms. There has been work [16] also on applying adaptive histogram techniques to Open Data on semantic Web.

We noticed that there were cases when using these techniques where optimizer produced a really bad plan that had orders of magnitude larger cost than the optimal one. These cases are not seen often and this is the reason why adaptive histograms can not deal with them. Although there are rare cases they have large impact to the histogram's estimations. But these estimations are provided in order to choose the best execution plan. So these bad estimations lead to bad execution plans and consequently large execution time.

Therefore the contribution of this thesis is that we detected such cases and created a method to make histogram provide the optimizer better statistics on these cases. Even though in some cases our method made worse estimations than the original statistic's provider, in the worst case scenarios, where the cost of the original plan was orders of magnitude larger than the optimal, our method achieved to change optimizer's plan and

avoid this large cost.

1.1 Outline

The remainder of this thesis is organized as follows:

- In Chapter 2, we refer the background and related work to histograms,query optimization and semantic web.
- In Chapter 3, we present our method.
- In Chapter 4, we present how our histogram's estimations will be used by the optimizer.
- In Chapter 5, we present the experiments and results of this thesis.
- In Chapter 6, we discuss some conclusions and what can be done as future work.

2. BACKGROUND

2.1 Histograms

A *histogram* [7] is a special type of column statistic that provides more detailed information about the data distribution in a table column. A histogram on an attribute X is constructed by partitioning the data distribution of X into $\beta(\geq 1)$ mutually disjoint subsets called buckets and approximating the frequencies and values in each bucket in some common fashion. This is the simplest form of a histogram.Database histograms have been studied extensively since their fist appearance in literature. So there have been a lot of different approaches to make them better in terms of consistency and at the same time keeping them efficient.

Histograms can be constructed by scanning the database tables and aggregating the values of the attributes in the table and similarly maintained in the face of database updates. However, large tables and usually updated data make these methods (proactive methods) of building histograms through a data scan really inefficient. Such histograms need to be periodically rebuilt in order to remain consistent after database updates, and this is exacerbating the overhead of these methods. So another approach to histogram construction that has been examined consists of query feed-back mechanisms that take into account actual sizes of query results to dynamically modify histograms so that their estimates are closer to reality.The main assumption behind self-tuning histograms is that, given enough query results to learn,they will be able to accurately capture the underlying data distribution. In addition to their dynamic nature,one key advantage of these approaches is also the low cost. Despite their attractive features, self-tuning approaches have also several disadvantages. As described by Khachatryan [9] changing the order of the learning queries can have a significant impact on the histogram precision. In 1 we can see that even in a 2-dimensional histogram when setting bucket limit to two buckets we get very different histograms after changing the order of the queries.Each row of the figure shows a sequence, the left column of the figure shows the order in which the queries arrive (denoted by numbers), the middle column is the situation after both queries are executed and buckets are drilled and the right column is the final configuration after one bucket is removed to meet the 2-bucket budget. This phenomenon is called *sensitivity to learning*.

An associated problem is that self-tuning methods struggle to capture complex data correlations in high-dimensional spaces. The importance of this disadvantage can be easily understood by the example presented by Lohman [11] and took place in a commercial system as DB2 where the original query executed in some seconds and the same query with one redundant predicate needed an hour to execute because the extra predicate totally threw off the optimizer. More specifically the column the extra predicate was on, was a composite key constructed of the first four letters of the policy-holder's last name, the first and middle initials, the zip code, and last four digits of his/her Social Security Number. But the original query had predicates on all those columns. And how many rows were there in the table? Ten million. So the predicate was completely redundant of the others, and

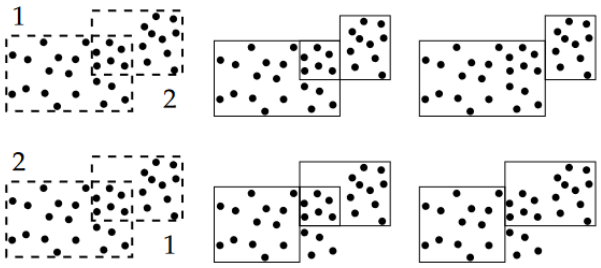

Figure 1: Two queries and the histograms corresponding to them. Each row represents a different query order.

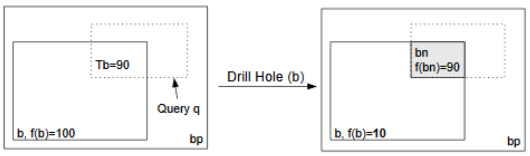

Figure 2: Drilling a hole in bucket *b* to improve histogram's quality.

its selectivity, $1/10^7$, when multiplied by the others, underestimated the cardinality by 7 orders of magnitude!!

One of the main representatives of self-tuning histograms is the *STHoles* [4]. STHoles in contrast of previous histogram construction approaches allows buckets to overlap. This characteristic makes STHoles to exploit query-feedback in a trully multi-dimensional way. STHoles allows for inclusion relationships between buckets. Therefore, an STHoles histogram results in a tree structure, where each node represents a bucket.A hole is a subregion of a bucket with different tuple density and is also a bucket itself. If a query result partially intersects with a bucket,then histogram may be refined by drilling new holes whenever the query results have much difference from histogram's prediction. Figure 2 shows a bucket b with frequency $f(b) = 100$. Suppose that from the result stream for a query *q* we count that $T_b = 90$ tuples lie in the part of bucket b that is touched by query $q, q \cap b$. Using this information, we can deduce that bucket b is significantly skewed, since 90% of its tuples are located in a small fraction of its volume. We can improve the accuracy of the histogram if we create a new bucket b_n by 'drilling' a hole in b that corresponds to the region $q \cap b$ and adjust b and $b'_n s$ frequencies accordingly as illustrated in Figure 2. So opening a new hole for the part of the bucket that partially intersects with the query solves the problem of different tuple density in the same bucket.

However drilling new holes increases the information stored into the histogram so more space is needed to store it. But keeping histogram efficient requires space limitations so when maximum size is reached some buckets must be merged according to one evaluation function which decides which buckets are less important or inaccurate. There are two main families of merges for STHoles histograms : *parent-child* merges and *sibling-sibling* merges.As said previously STHoles histograms result to a tree structure so the fisrt family

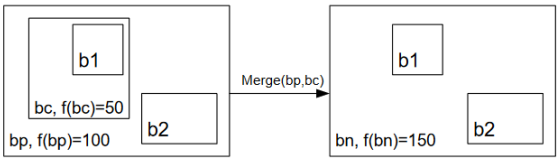

Figure 3: Parent-Child Merge.

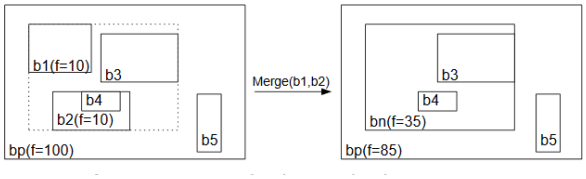

Figure 4: Sibling-Sibling Merge.

of merges describes the situation when a bucket is merged with its parent and the second one the situation where two buckets with the same parent are merged possibly taking some of the parent's space.In Figures 3 , 4 you can see two examples for both of them.

Moerkotte [12] defined the *q-error* to measure deviations of size estimates from actual sizes. He derived two very important results based on the q-error. (1) There are bounds such that if the q-error is smaller than this bound, the query optimizer constructs an optimal plan. (2) If the q-error is bounded by a number *q*, it is proved that the cost of the produced plan is at most a factor of q^4 worse than the optimal plan.

Another recent approach by Anagnostopoulos [1] introduced a query-driven function estimation model of analyst-defined data subspace cardinality. The proposed estimation model is highly accurate in terms of prediction and accommodating the well known selection queries such as multi- dimensional range queries. The function estimation model quantizes the vectorial query space, by learning the access patterns over a data space.After the learning-training part it associates query vectors with their corresponding cardinalities of the defined data subspaces according to their Euclidean distance (2-norm) and abstracts and employs query vectorial similarity to predict the cardinality of an unseen/unexplored data subspace.It also identifies and adapts to possible changes of the query sub-spaces based on the theory of optimal stopping.

2.2 Query Optimization

As discussed previously query optimizers rely on histograms and their data distribution estimations to be able to apply their cost models and choose the optimal or an efficient one between the different query plans. But query optimizers have to balance between overhead to the query execution, and optimality of produced query plans. But how are they doing it?

Lets see the path that a query traverses through a DBMS until its answer is generated [8]. First of all the *Query Parser* checks the validity of the query and then translates it into an internal form usually a relational calculus expression or something equivalent. After that the *Query Optimizer* examines all algebraic expressions that are equivalent to the given query and chooses the one that is estimated to be the cheapest. The *Code Generator* or the Interpreter transforms the access plan generated by the optimizer into calls to the query processor. And in the end the *Query Processor* actually executes the query.

So query optimizer has to search in the space of all equivalent algebraic expressions and choose the cheapest one according to its *Cost Model* and the *Size-Distribution Estimator* (e.g. Histograms). In practice this is not true because typical query optimizers make some restrictions to reduce the space they explore.After adding these restrictions optimizer has to search for the optimal plan.The most important strategy used by most commercial systems to do this is *Dynamic Programming*. Although there have been other interesting approaches based on randomized algorithms and other search strategies.

There has been a lot of research and effort on making histograms as consistent as possible [14] and query optimizers to produce effective and as more as possible optimal query plans keeping in mind the overhead limitations.Although query optimization even after so many years of research is not a ”solved” problem [11]. And the Achilles Heel of query optimization, is the estimation of the size of intermediate results,cardinalities.

2.3 Semantic Web

The Resource Description Framework (RDF) [10] is a W3C Recommendation for the formulation of metadata on the World Wide Web. RDF Schema [3] (RDFS) extends this standard with the means to specify domain vocabulary and object structures. These techniques enable the enrichment of the Web with machine-processable semantics, thus giving rise to what has been dubbed the semantic Web.

The basic building block in RDF is an subject-predicate-object triple, commonly written as P(S,O). That is, a subject S has a predicate (or property) P with value O. Another way to think of this relationship is as a labeled edge between two nodes: [S] − P → [O].

RDF Schema is a mechanism that lets developers define a particular vocabulary for RDF data (such as the predicate hasWritten) and specify the kinds of objects to which predicates can be applied (such as the class Writer). RDFS does this by pre-specifying some terminology, such as Class , subClassOf and Property , which can then be used in application-specific schemata. RDFS expressions are also valid RDF expressions – in fact, the only difference with ‘normal’ RDF expressions is that in RDFS an agreement is made on the semantics of certain terms and thus on the interpretation of certain statements. For example, the subClassOf property allows the developer to specify the hierarchical organization of classes. Objects can be declared to be instances of these classes using the type property. Constraints on the use of properties can be specified using domain and range constructs.

SPARQL [13] is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format.

In SQL relational database terms, RDF data can also be considered a table with three columns – the subject column, the predicate column, and the object column. The subject in RDF is analogous to an entity in a SQL database, where the data elements (or fields) for a given business object are placed in multiple columns, sometimes spread across more than one table, and identified by a unique key. In RDF, those fields are instead represented as separate predicate/object rows sharing the same subject, often the same unique key, with the predicate being analogous to the column name and the object the actual data. Unlike relational databases, the object column is heterogeneous: the per-cell data type is usually implied (or specified in the ontology) by the predicate value. Also unlike SQL, RDF can have multiple entries per predicate; for instance, one could have multiple 'child' entries for a single 'person', and can return collections of such objects, like 'children'.

Thus, SPARQL provides a full set of analytic query operations such as JOIN, SORT, AGGREGATE for data whose schema is intrinsically part of the data rather than requiring a separate schema definition. However, schema information (the ontology) is often provided externally, to allow joining of different datasets unambiguously. In addition, SPARQL provides specific graph traversal syntax for data that can be thought of as a graph.

SemaGrow [5] is a successful federated SPARQL query processing system that relies on metadata about the data sources to produce an efficient query plan and achieves good results while introducing little overhead to the query execution. It uses Dynamic Programming to discover the ”best” among the query plans.

2.4 Histograms and Semantic Web

Most work on histograms has focused on approximating numeric values, in one or multiple dimensions (attributes). Instead of statically rescanning database tables,STHoles and in general workload-aware self-tuning histograms have introduced little overhead and have been successfully used in relational database tables. These techniques because of their query-driven construction result to histograms that are focused to the workload and the more frequently queried data regions will provide more accurate statistics. Furthermore, they are able to adapt to changes in data distribution and thus are well-suited for datasets with frequently changing contents.They are, however, for the most part targeting numerical attributes, since they exploit the idea that a value range is an indication of the size of the range.

But *URIs* the most prominent data type of semantic web can't be defined using numeric values and ranges, so there has been presented [16] an algorithm for building and maintaining multi-dimensional histograms based on STHoles but extended to be able to handle arbitrary data types. There have been defined also two ways to handle URIs either using prefix ranges or using the Jaro-Winkler [15] metric. Prefixes can naturally express ranges

of semantically related resources given the natural tendency to group together relevant items in hierarchical structures such as pathnames and URIs. The Jaro-Winkler similarity is used to define the distance between two strings so it is suitable for URI comparison since it provides preference to the strings that match exactly at the beginning.

3. HISTOGRAMS

At this chapter we are going to present our method. More specifically, we present our method during the histogram construction and maintenance and also during the cardinality estimation. We also use an example taken from the experiments of chapter 5 to explain the way our method is going to achieve better performance.

3.1 Our Method

We added one extra value maintained by histogram in order to make it more robust. More specifically in every bucket except keeping only the frequency (amount of all tuples contained in the bucket) and the distinct values in each dimension which are required to find the median value to make the estimation, we now also keep one maximum number per dimension which is the maximum number of tuples that one distinct value can match. For example consider that we have a table with employees and their department and a histogram describing it. There is a bucket containing 3 departments (1,2,3) and 1200 employees that everyone of them works in just one department. The estimation of the histogram for this bucket when asked how many employees work in department number 1 would be 400 each. But actually department 1 has 100 employees department 2 100 and department 3 1000. So our additions in this 2-dimensional bucket would be one number for maximum number of tuples mathing a department which is equal to 1000 and maximum number of tuples matching an employee which is equal to 1.

3.2 Definition of Our Method During Histogram Construction and Maintenance

So in this section we describe how these values are constructed and how we maintain them when the histogram is updated.

Drilling a Hole

Maximum Triples per Subject = Max{For each distinct subject return the count of all the tuples in the resultset that match with this subject}
Maximum Triples per Predicate = Max{For each distinct predicate return the count of all the tuples in the resultset that match with this predicate}
Maximum Triples per Object = Max{For each distinct object return the count of all the tuples in the resultset that match with this object}

Merge (Sibling-Sibling or Parent-Child)
When two buckets are merged the way we compute the above values of the merged bucket is described below where b1 and b2 are the buckets to be merged:

Maximum Triples per Subject of the merged bucket = $Max\{MTpS_{b1}, MTpS_{b2}\}$
Maximum Triples per Predicate of the merged bucket = $Max\{MTpP_{b1}, MTpP_{b2}\}$
Maximum Triples per Object of the merged bucket = $Max\{MTpO_{b1}, MTpO_{b2}\}$

, where *MTpS* means the Maximum Triples per Subject, *MTpP* Maximum Triples per Predicate and *MTpO* Maximum Triples per Object.

3.3 Definition of Our Method During Cardinality Estimation

We are going to use the values that we introduces before in histogram's estimation. To do so we introduced the following 3 metrics so as to determine the effect of the maximum values at histogram's estimation in some different ways.

$$K_1 = currentEstimation + (Constant/100) * Max$$
$$K_2 = log(Max/currentEstimation) * Max * \lambda$$

At fisrt metric K1 we have a parameter defined by user which controls the overestimation that will be done. This has the advantage that we can manually control our overestimation so as to reduce that when we are confident about our estimations and adjust it when we think our histogram is not so accurate. But its disadvantage is that because it is a parameter we will have same approach at all occasions even when we do not want to make a large overestimation.

The second K2 is the one we believe that will produce the best results. The reason why we believe this is that because of the logarithm properties we are going to make an overestimation depending on the difference between the current estimation and the maximum and not just a constant as in metric number K1. So this is a way to dynamically control the overestimation and not dominate the current estimation which may have negative impact in the general case.

Table 1: Table with actual cardinalities

Query	Cardinality
$P1$	287278
$P2$	69216
$P3$	20284
$P1 \bowtie P2$	32252
$P2 \bowtie P3$	5277
$P1 \bowtie P3$	8695
$P1 \bowtie P2 \bowtie P3$	2036

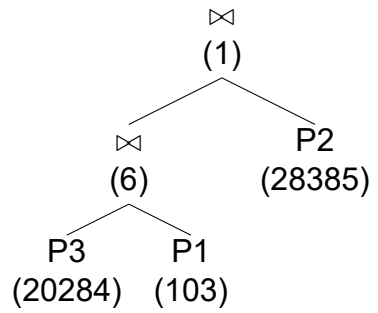

Figure 5: Produced Plan with current estimation.

3.4 Our Method's Advantages

The following SPARQL query consists of a join of three patterns at the same attribute which we extracted from our experiments section that follows to show this case that is exactly the case where our method achieves a lot better performance than the previous one.The actual cardinality of the result of this query is 2036 tuples. The reason we picked this query is because as we can see from the statistics (real statistics provided by training the histogram with version 3.2 of dbpedia datasets),the bucket containing the statistics for query P1 has a very large number as maxTriplesPerObject compared to all the Triples of bucket. And the value of the object for which we get the maximum tuples is the one we use on the query so we have an edge case in which we want to show the advantages of our method.

Listing 3.1: The query

```
SELECT ?name WHERE {
  ?s skos:subject <http://dbpedia.org/resource/Category:Living_people>. (P1)
  ?s rdf:type dbo:Artist. (P2)
  ?s foaf:name ?name. (P3)
}
```

Table 2: Table with histogram statistics

Query	Triples	Distinct Subjects	Distinct Objects	Maximum Triples per Subject	Maximum Triples per Object
P1	363368	349776	3511	8	287278
P2	397393	369000	14	4	128614
P3	20284	20284	9351	1	385

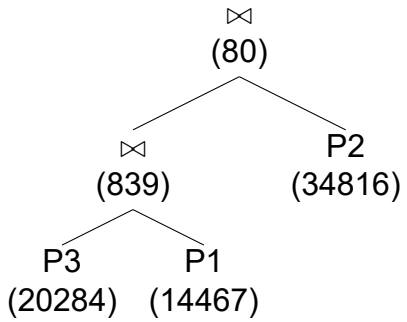

Figure 6: Produced Plan when constant=5.

3.4.1 Insufficient Learning and Unlimited space

At Tables 1 and 2 we can see the actual cardinalities of the corresponding queries and the statistics of the enclosing bucket of each one of them.

At Figure 5 we can see the plan produced by the optimizer when provided with the original estimations from histogram.The estimation for P1 is 103 and is calculated from the table with statistics by the type $Triples/DistinctObjects$ but as mentioned before the actual cardinality is 287278 so this fault affects optimizer's decision and keeps it at the top of the plan.And also affects the estimated cardinality of the whole plan. As we can see estimated cardinality of the whole plan is equal to 1 but the actual one is equal to 2036.Our goal is by using our method make optimizer to change the selected plan and experiment which one of our metrics is better in each case and which constant seems to have better performance for each of them.

3.4.1.1 Metric K1

At this experiment we are going to show how K1 metric behave with different values for our constant. More specifically we are going to use the values 5,10,20 and 50 for our constant.

At Figure 6 we can see that as we expected very small values such as 5 had bad performance and they didn't even affect the selected plan.

So we can not expect such small values to have an impact to optimizer's plan.

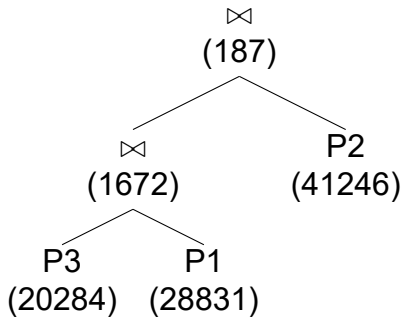

Figure 7: Produced Plan when constant=10.

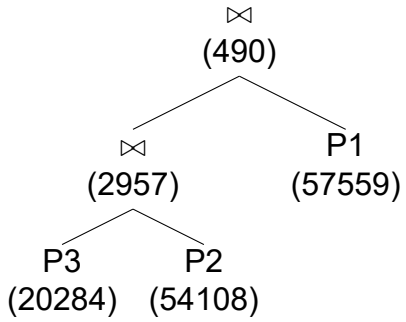

Figure 8: Produced Plan when constant=20.

For the case where constant is equal to 10 we can see (Figure 7) that even though we can notice that our estimations are a lot bigger than the previous ones they didn't manage to affect optimizer's plan because P2's cardinality still remains bigger than P1's. This is a case where we expect the metric K2 to have better performance because the $log(Max/currentEstimation)$ in the case of P2 will be significantly smaller than in the case of P1.

But for the cases where constant is equal to 20 (Figure 8) we can see that even though the final estimation is smaller than the actual cardinality our intermediate estimations affected the produced plan.

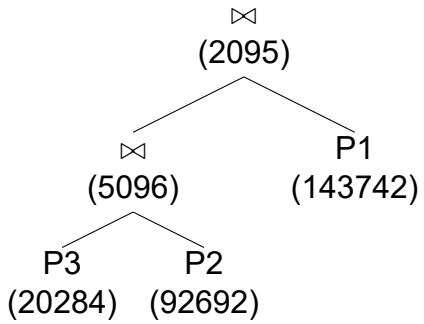

Figure 9: Produced Plan when constant=50.

Finally we can see that for the case where constant is equal to 50 the estimated cardinality is very close to the actual one for this edge case. But in the average case this huge overestimation which dominates the previous ‘average’ estimation may affect optimizer to take non-optimal decisions.

To conclude we can see that at this case this metric behaves better with big values of the constant because uncertainty for data is big but in the average case this large overestimation may result to bad estimations and non-optimal plans.

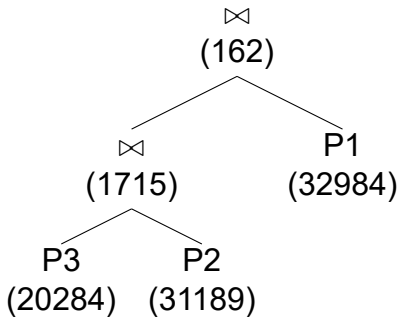

Figure 10: Produced Plan when $\lambda = (log(2) * 100)^{-1}$.

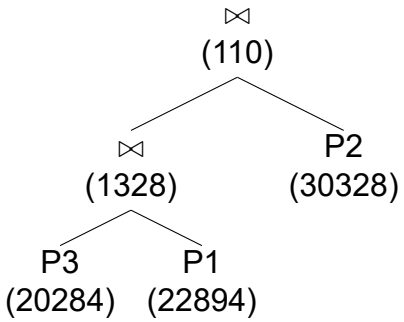

Figure 11: Produced Plan when $\lambda = (log(e) * 100)^{-1}$.

3.4.1.2 Metric K2

At this case we expect from metric K2 an overestimation which will affect the selected plan from optimizer but it will not also make a huge difference between the previous estimation and the estimation after evaluation of metric K2.

As we can see in Figures 10, using our addition although our estimation for P1 still remains inaccurate, it is a lot larger than the previous one. So the new estimation is more accurate than the previous one.

This fact changes optimizer's plan decision and optimizer prefers to make the inner join with the more 'confident' P2 and P3 and not P1 which can have a very large cardinality exactly what we wanted to achieve. Also we can notice that the final estimation for the whole query is very larger than the previous estimation but still remains inaccurate cause we are in a rare edge case senario where the actual estimation for P1 is orders of magnitude bigger than the average of the bucket.

But in Figures 11,12 we can notice that the plan is the same as the selected plan with the previous estimations when λ is bigger. This is because P1 cardinality is estimated to be 10001 which is smaller than estimated cardinalities of P2 and P3 as it is in the original estimations so P1 is selected to be at the inner join of the plan which is a non-optimal plan.

So from this experiment we can conclude that metric K1 has better performance at edge cases like the one selected here with a bigger number as λ and this is something we

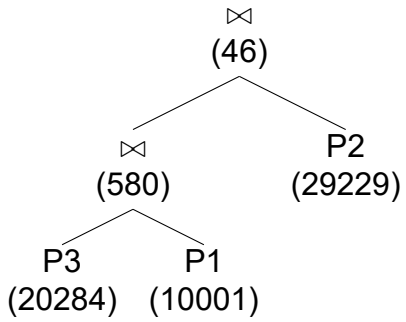

Figure 12: Produced Plan when $\lambda = (log(10) * 100)^{-1}$.

expected because bigger λ means that will affect more the estimation.

But in general we can see that metric K2 achieves to affect optimizer's decision and at the same time making a logical overestimation without dominating the previous estimation. This means that even though our cardinality estimation for the whole query is smaller than the actual cardinality in this case,in the average case it will be very robust.

Table 3: Table with histogram statistics

Query	Triples	Distinct Subjects	Distinct Objects	Maximum Triples per Subject	Maximum Triples per Object
P1	508696	292005	1179	18	287278
P2	508696	292005	1179	18	287278
P3	20284	20284	9351	1	385

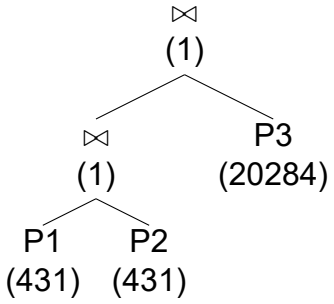

Figure 13: Produced Plan with current estimation.

3.4.2 Insufficient Learning and Limited Space

You can see the histogram statistics for this case at Table 3.

Table 3 shows us that query P1 and P2 happen to fall in the same bucket.This is very interesting because we are going to see how our method and every metric separately behaves in such cases where one merged bucket contains very ‘different’ data.

At Figure 13 we can see the plan produced with current estimation.

3.4.2.1 Metric K1

As in the previous experiment we expect again to have a better performance from bigger values of the parameter.

Again as we expected very small values such as 5 had bad performance and they didn’t even affect the selected plan.

At Figures 15,16 we can see that for these mid-range values this metric achieves to changes optimizer’s plan while not dominating the original estimations. But because queries P1 and P2 happen to fall in the same bucket the decision to keep query P1 at the root of the selected plan and not into the inner join is made by luck and this is something we cannot avoid with our method.

But in the case where constant is equal to 50 ,Figure 17 , the overestimated cardinality even if for query P1 still is not enough,for query P2 is a lot larger and this is one case

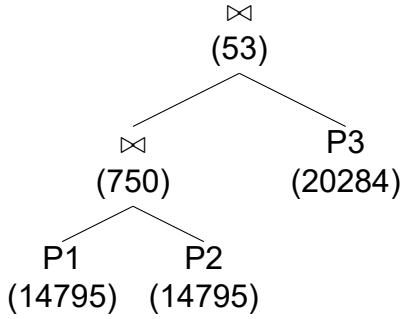

Figure 14: Produced Plan when constant=5.

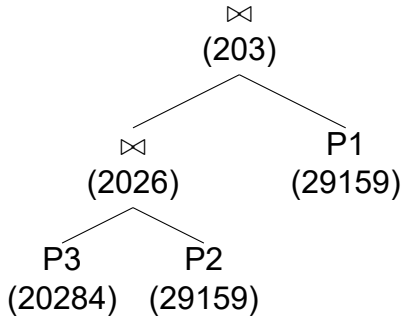

Figure 15: Produced Plan when constant=10.

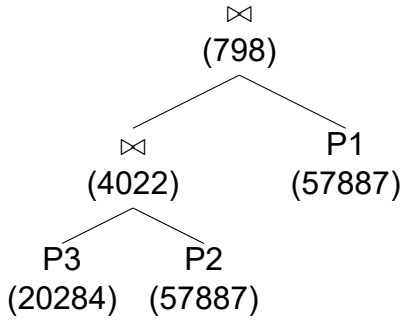

Figure 16: Produced Plan with K2 and constant = 20.

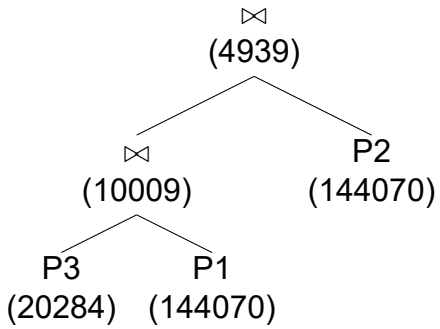

Figure 17: Produced Plan with K2 and constant = 50.

where our method could have negative impact to histogram's estimation and optimizer's plan selection. Because of the multiple merges there might be a very large outlier which ,especially when using metric K1 with a big constant, will dominate the estimation and produce really big values that are very larger than the actual cardinalities for the average case.

Finally,we can conclude that in this experiment we saw the case where our method and especially metric K1 can have negative impact to the histogram's estimations. Metric K1 behaves better with mid-range values around 10-20 set to the constant and more than this can dominate the estimations when there is a bucket with a large difference between the maximum and the average value.

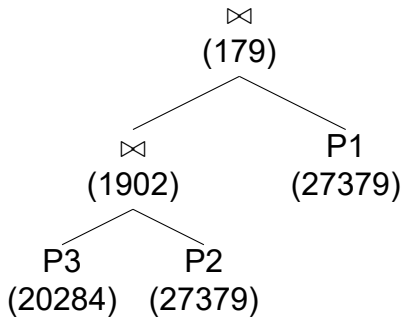

Figure 18: Produced Plan when $\lambda = (log(2) * 100)^{-1}$.

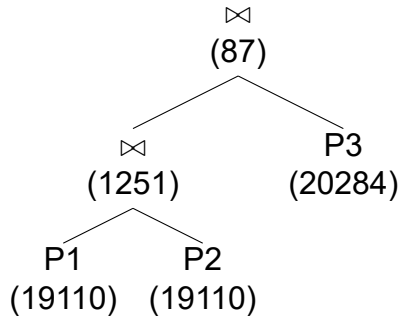

Figure 19: Produced Plan when $\lambda = (log(e) * 100)^{-1}$.

3.4.2.2 Metric K2

As said previously we expect from metric K2 an overestimation which will affect the selected plan from optimizer but it will not dominate the previous estimations. At this case we expect this metric to have better results because queries P1 and P2 fall in the same bucket while actual cardinality of P1 is a lot larger than P2's so this metric will keep the overestimation closer to the average than the metric K1 where a big value of the constant will make P2's estimated cardinality a lot larger than the previous one.

As we can see in Figure 18 using a bigger value for λ achieved again to change the plan selected by the optimizer even though the estimated cardinalities are still smaller than the actual ones. So as seen in the previous experiment this metric seems to balance successfully between the edge case and the average case because it can overestimate enough to change optimizer's plan but also keep the estimation close to the average.

At Figures 19, 20 we notice that there is no change on the plan at these cases.

For the case where $\lambda = (log(e) * 100)^{-1}$ we notice that it is very close to achieve to change the plan because the estimated cardinality for P1 and P2 is 19110 and if it was more than 20284 it would that is te estimated cardinality of P3 then there would be a change to optimizer's plan. So we can consider this value of λ as a good one that in cases where we trust our histogram's estimations we can prefer this value against the first one.

For the case where $\lambda = (log(e) * 100)^{-1}$ the over estimation is a lot smaller than the pre-

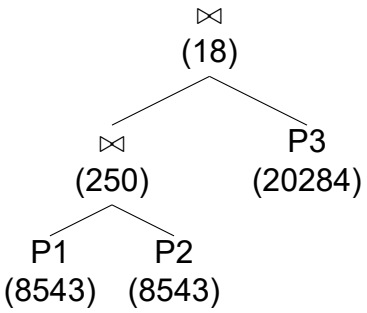

Figure 20: Produced Plan when $\lambda = (log(2) * 100)^{-1}$.

vious ones and of course it did not manage to affect the plan.

So as mentioned before metric K2 seems to have good performance edge cases like this with a bigger number as λ. But we also noticed that it can dynamically keep te overestimation closer to the average case when compared with K1 which is a user defined constant which at some cases may dominate the average estimation and this may lead to really bad decisions from the optimizer.

3.5 Conclusion

So concluding,at this chapter we presented our method. We saw that we compute and maintain some extra fields in every bucket of the histogram. These fields are related to the biggest outliers that the bucket includes and are used during the cardinality estimation by the two metrics that we defined. These metrics use also one parameter each that can be assigned by the user in order to choose how much impact he wants our method to have at the final estimation. We also described an example where our method was used and had a lot better performance than the previous one and using this example we presented some advantages and disadvantages of our method and our expectations from each one of the two metrics that we defined.

4. OPTIMIZATION

In this chapter we are going to describe how optimizers and specifically how the optimizer that we used to test our method uses the statistics provided by the histogram or other sources in order to choose the optimal plan and execute the queries.

4.1 Execution Plan Construction

First, the query string is parsed into a tree where the leaves correspond to triple patterns and the intermediate nodes represent relational operators such as join,union,etc. Query optimization restructures and augments query parse trees into execution plans where (a) besides the retrieval of triples matching a pattern,leafs can also represent complex subqueries to be executed by an endpoint; and (b) the order or the execution is changed.

Execution plan construction is generally approached as a search through the space of possible plans guided by a cost function that estimates the efficiency of each solution.

The cost of complex expressions is estimated recursively using a cost model over statistics about its subexpressions. Different cost models are defined for each operator,for example,the Union operator sums the costs of its subexpressions.

4.2 Statistics Provider

The statistics are provided by the histogram and include the cardinality of the subexpressions' results, as well as the number of distinct subjects, predicates, and objects appearing in these results. The number of distinct entities is used to estimate the selectivity of JOIN and FILTER expressions: the ratio of the number of tuples that satisfy the expression over the number of tuples in the complete relation. First of all we have to see how the cardinality estimation of an expression is defined. The cardinality can be estimated using the source metadata maintained by the Resource Discovery component. Given a triple pattern the Resource Discovery component will return a list of candidate data sources that is believed may contain matching triples along with statistics about this data sources. These statistics is:

1. The estimated number of the triples that match the triple pattern in the given data source.
2. The estimated distinct subjects matching the triple pattern.
3. The estimated distinct predicates matching the triple pattern.
4. The estimated distinct objects matching the triple pattern.

5. The estimated maximum triples per subject/predicate/object as described in the previous chapter.

The cardinality of a triple pattern is provided by the histogram. On the other hand, for more complex expressions it is necessary to make an estimation based on our available statistics. First, it is necessary to define the notion of the selectivity factor. Given a generic predicate *p* and a relation *R* the selectivity factor is defined as the ratio of the number of the tuple that satisfy predicate *p* over the total cardinality of *R*. By estimating this factor can be estimated the cardinalities of more complex expressions such as joins. For semagrow's approach they distinguish two kinds of selectivity factors: the join selectivity that is applied to join expressions and the filter selectivity that is applied to filter expressions.

4.3 Join Selectivity Estimation

The Join Selectivity estimation JoinSel(E1 join E2) of a join expression E is defined as follows:

- $JoinSel(E1 \bowtie E2) = min(JoinSel(E1), JoinSel(E2))$
- $JoinSel(T) = min(\frac{1}{d_i})$

where E1 and E2 are any pattern or join expression, T is a simple triple pattern, and d_i is the number of distinct values for the i_{st} join attribute in triple pattern T. Note that for the computation of the join selectivity we assume:

- *Independence* between the join arguments.
- *Containment* of the smallest (in number of tuples that satisfy it) pattern into the other,so that selectivity is determined by the number of distinct values.
- *Uniformity* in the distribution of distinct values, so that for a given pattern, the specific values for the positions in the pattern (subject, predicate, or object) that are binded do not influence the computation.

It is obvious that these assumptions do not hold in the general case. Therefore, the success of optimization strategies depends on the quality of the metadata. Specifically, it depends on the availability of distinct values counts for subsets of the complete dataset, where these subsets partition the dataset in such a way th at the assumptions above hold.

4.4 Filter Selectivity Estimation

The Filter Selectivity estimation Sel(C , E) of a filter expression (E FILTER C) is defined as follows:

- $Sel(C1\ AND\ C2, E) = Sel(C1, E) * Sel(C2, E)$
- $Sel(C1\ OR\ C2, E) = max(Sel(C1, E), Sel(C2, E))$
- $Sel(NOT\ C, E) = 1 - \ Sel(C, E)$
- $Sel(E) = \frac{1}{2}$.

4.5 Cardinality Estimation

We can now see how the cardinality estimation is defined:
Given a source S, the cardinality estimation Card(E , S) of an expression E is defined recursively as follows:

- $Card(spo, S) = Card_{spo}$ given by the Histogram
- $Card(E1 \bowtie E2, S) = Card(E1, S) * Card(E2, S) * JoinSel(E1 \bowtie E2)$
- $Card(E1\ optional\ E2, S) = Card(E1, S) * Card(E2, S) * JoinSel(E1 \bowtie E2)$–$Card(E1) * Card(E2) * (1 - \ JoinSel(E1 \bowtie 2)$
- $Card(E1\ union\ E2, S) = Card(E1, S) + Card(E2, S)$
- $Card(E\ filter\ R, S) = Card(E, S) * Sel(R, E)$

We write Card(E) to denote the sum of the cardinality estimations over all the candidate data sources for expression E.

The cost of each expression is evaluated based on the cost estimates for the individual operands. Since there are various approaches on defined cost functions semagrow uses a traditional formulas in distributed databases. This simple cost function models the overall effort spent to compute the expression under consideration. Let the constants $C_{transfer}$ and $C_{process}$ be the communication and computation unit cost for a single tuple respectively. Moreover, C_{query} is the cost involving the execution of a query to a remote source. These constants can be given by the system configuration and/or on a data source basis.

4.6 Cost Estimation

The cost function Cost(E) of an expression E is defined recursively as follows:

- $Cost(spo) = Card(spo) * C_{process}$
- $Cost(E1\ mergejoin\ E2) = Cost(E1) + Cost(E2) + (Card(E1) + Card(E2)) * C_{process}$

- $Cost(E1\ hashjoin\ E2) = Cost(E1) + Cost(E2) + Card(E1) * C_{process} + Card(E1 \bowtie E2) * C_{process}$
- $Cost(E1\ bindjoin\ E2) = Cost(E1) + C_{transfer} * Card(E1 \bowtie E2) + C_{process} * Card(E1) + C_{query} * (Card(E1)/batch)$ where batch is the number of bindings of E1 dispatched to the source of E2 by a single query , so that Card(E1)/ batch is the number of queries needed in order to dispatch all bindings.
- $Cost(E1\ union\ E2) = Cost(E1) + Cost(E2)$
- $Cost(E\ filter\ R) = Cost(E) + C_{process} * Cost(E)$
- $Cost(Sort(E)) = Cost(E) + C_{process} * Card(E) * log(Card(E))$
- $Cost(SourceQuery(Q)) = C_{query} + C_{transfer} * Card(Q, Site(Q))$.

Note that the SPARQL operators for which the cost function is defined above, covers all expressions for which meaningful optimization can be performed by the planner. SPARQL constructs, such as GROUP, that are not covered are left as they appear in the original query, although inner expressions will get optimized. The planner optimizes the maximal subqueries that are fully covered, leaving the relationship between such subqueries outside the scope of optimization. For example, in an expression such GROUP G (T1, T2) the inner join T1 join T2 might get reordered although the cost of the overall expression is not defined and the relationship of GROUP G (T1, T2) to the rest of the query will not be altered. Note also that, in contrast to the cardinality estimation function, the cost function is dependent on the specific join algorithm. It is assumed that the decomposer can select between bind join, merge join and hash join each having a different cost for given E1, E2. For example , consider the Bind join operator. An intuitive description of the evaluation of (E1 bind join E2) is: The execution engine first evaluates E1 , and the results of E1 will be served as bindings for the evaluation of the E2. If the evaluation of E1 yields n different tuples, then the evaluation of E2 will be performed n times, each time with a different binding. It should be noted here that certain optimizations can be applied in a bind join implementation that will group a set of bindings into a single batch evaluation. Therefore, if the batch size is b, the actual number of the E2 evaluation is reduced to n/b. Since its evaluation of E2 results to a separate query to the underlying data source it is easy to verify that E1 with smaller cardinalities will result to a lower cost expression. This intuitive observation is depicted to the cost formula of bind join. Notice that the cost of the bind join is asymmetric, namely the cost of (E1 bind join E2) can differs vastly from the cost of (E2 bind join E1). On the other hand, the cost of merge join is symmetric since the algorithm must evaluate both E1 and E2 independently.

4.7 Query Planning

With cost estimations defined, it is possible to proceed to evaluate different plans in order to identify the one that is optimal with respect to the cost model. Dynamic programming is

used to do this, the standard enumeration algorithm for join ordering optimization that has been used successfully in many database systems. It takes as input a basic pattern group, i.e. a part of the algebraic tree of a query that consists only of joined triple patterns and optionally a set of filters. It returns as output a potential reordering of the join operators and each operator is annotated with the specific algorithm to be used.

4.8 Conclusion

So in this chapter it is shown the way our histogram's estimations are going to be used from the optimizer that we are going to use to evaluate our method.It is clear that both the estimations provided and the cost model play a very significant role to the optimizer's decision of the execution plan.

5. EXPERIMENTS

At this chapter we present the experiments that took place in order to evaluate our method. Then we present and analyze the results that the above experiments led us to.

5.1 Experimental Setup

We installed latest version of Virtuoso Triple Store [6] from this link `http://vos.openlinksw.com/owiki/wiki/VOS/VOSDownload` on machine running Ubuntu Linux 16.04. After that we loaded into Virtuoso the required datasets for our queries from DBpedia [2] version 3.2 from the following link `http://wiki.dbpedia.org/data-set-32`. Finally we implemented our additions on the open-source STRHist [16] histogram at this link `https://github.com/lefteriskat/strHist` and used it for our experiments. And the optimizer we used is Semagrow [5] which is also open-source and can be found here `https://github.com/semagrow/semagrow`.

5.2 Workload

Before executing the experiments we had to find a workload which consists of SPARQL queries that can be uses in real world cases. So we created 3 templates. The arguments of the query named as '{}' are the arguments where we are going to place constants.

At the first one we are querying the endpoint to get all results by their names and associated article categories that have to do with a specific type of dbpedia ontology. As an example of this query we can get the at the position of the '{}' the <dbo:Artist> URI which means we are going to get all the resources of dbpedia that have to do with artists and by their names and associated categories that the resources belong to.

Listing 5.1: First Template

```
SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type {} .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
```

At the second template we are querying the endpoint in order to get Film names and the categories that they belong to and we filter the results according to the release date of the films.So as an example we can have in the position of '{_op}' the >= operator and in the position of the '2010-01-01'8sd:date which means that we are going to receive all the films that are released since 2010.

Listing 5.2: Second Template

```
SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
} FILTER (releaseDate {_op} {})
```

Finally the third template consists of different and more complicated queries that can be described with the following template where the patterns that are in parentheses are optional and not used in all the queries.

Listing 5.3: Third Template

```
SELECT * FROM <http://dbpedia3.2.org> WHERE {
        (?s  rdf:type {} .)
        ?s foaf:name ?name.
        ?s skos:subject {}.
        (?s dbo:releaseDate ?releaseDate.)
}
```

5.3 Results and Analysis

At this section we are going to present and analyze the results using the above workload. We are going to evaluate our method based on two factors.

The main factor is if our method achieves to make optimizer select better plan than the one selected when optimizer is provided with the original histogram's estimation.This is done by comparing the plan selected by the optimizer when provided with each one of the metrics defined by our method or the original estimation against the plan selected when optimizer is provided with 100% accurate statistics. So let's assume that the plan selected when optimizer is provided with the 100% accuarate statistics is the optimal one. If the plan selected when the original estimation is provided is different from the optimal one but when provided with some metric of our method the plan is equal to the optimal we achieve our goal and our method has a positive impact. To the opposite if the plan selected when the original estimation is provided is equal to the optimal one but when provided with some metric of our method the plan is different from the optimal, our method has a negative impact. And when the original estimation and our method's estimation are the same either they are identical to the optimal or not there is no impact from our method.

The other factor is the accuracy of the cardinality estimation when using our method compared with the original estimation,the actual result and the estimation made with the 100% accurate statistics. Specifically to evaluate this factor we use measure the average *absolute estimation error (ABS)* and the average *root mean square error (RMS)* of the original estimation and the estimations made with our method and compare the results.

Finally the experiments that follow are separated to two sections. At the first one we trained our histogram with 150 queries but we did not put any limit to the maximum number of

buckets. At the second one we trained our histogram with the same queries but we set the maximum bucket number limit to 30 in order to observe the difference in our method's results at these two cases.

5.3.1 Insufficient Learning and Unlimited Space

At tables 4 and 5 we can see the results we got after running the workload. These tables include the actual cardinalities at the Actual column,the cardinality estimated when histogram is provided with the actual statistics for every subquery at the BestEst column and the cardinalities estimated when our method is used. Especially as you can see we tested our method using both metric K1 and K2.

For metric K1 we assigned five different values to the constant in order to observe the difference between the results when eaxh of them is used. More specifically, Metrc1.1 refers to metric K1 when constant=5,Metric1.2 when constant=10,Metric1.3 when constant=20 and Metric1.4 when constant=50. For metric K2 we assigned three different values to the constant. More specifically, Metric2.1 refers to metric K2 when λ = 2, Metric2.2 when λ = e =2.71828... and Metric2.3 when λ = 10.

As we can see at tables 4 and 5 both the average ABS error and the average RMS error are very close if we compare our method's results with the previous estimations results.

5.3.1.1 Unlimited Results Analysis

Queries where we achieve better performance

QUERY 27

We analyzed in detail this case at section 3.

Queries where we achieve worse performance

QUERY 22

Listing 5.4: Query 22

```
SELECT * WHERE {
        ?s skos:subject <http://dbpedia.org/resource/Category:American_films>. (#P1)
        ?s rdf:type dbo:Film. (#P2)
        ?s foaf:name ?title. (#P3)
        ?s dbo:releaseDate ?releaseDate. (#P4)
}
```

This is an interesting query because if we look at the P1 and P2 subqueries carefully we can notice that P1's result actually is a subset of P2. So there independence assumption that is made through our cost model will lead us to really inaccurate estimations.

As we expected even though we provided histogram with the actual cardinalities of all the subqueries the estimated cardinality (Figure 21) of the whole query still remains a lot smaller than the actual one that is 9914 tuples.

Table 4: Results for metric K1 (Unlimited Space).

Query	Actual	BestEst	CurrEst	Metric1.1	Metric1.2	Metric1.3	Metric1.4
01.	381939	66348	33174 (T)	36475 (T)	39775 (T)	46376 (T)	66182 (T)
02.	46284	28356	16169 (T)	17585 (T)	19005 (T)	21841 (T)	30345 (T)
03.	46	15171	16587 (T)	19119 (T)	21654 (T)	26718 (T)	41918 (T)
04.	5402	4295	4295 (T)	4511 (T)	4724 (T)	5152 (T)	6441 (T)
05.	0	105	33174 (F)	36491 (F)	39808 (F)	46438 (F)	66332 (F)
06.	6067	6569	15557 (T)	16862 (T)	18164 (T)	20771 (T)	28595 (T)
07.	506	4472	4472 (T)	4907 (T)	5342 (T)	6209 (T)	8819 (T)
08.	13844	4720	5770 (T)	6111 (T)	6451 (T)	7134 (T)	9179 (T)
09.	6792	3468	3468 (T)	3641 (T)	3814 (T)	4161 (T)	5201 (T)
10.	8182	8619	8619 (T)	9051 (T)	9480 (T)	10343 (T)	12927 (T)
11.	103978	24994	3510 (F)	3860 (F)	4213 (F)	4917 (F)	7033 (F)
12.	95979	23640	3317 (F)	3648 (F)	3981 (F)	4645 (F)	6647 (F)
13.	83184	21625	3134 (F)	3451 (F)	3762 (F)	4393 (F)	6287 (F)
14.	63099	15528	2915 (F)	3209 (F)	3500 (F)	4086 (F)	5846 (F)
15.	25177	27	1152 (T)	1270 (T)	1384 (T)	1620 (T)	2326 (T)
16.	51030	3016	530 (T)	583 (T)	635 (T)	743 (T)	1060 (T)
17.	59039	4380	730 (F)	802 (F)	874 (F)	1021 (F)	1459 (F)
18.	71819	6487	985 (F)	1083 (F)	1181 (F)	1378 (F)	1970 (F)
19.	91908	13133	1636 (F)	1803 (F)	1963 (F)	2293 (F)	3281 (F)
20.	129826	27859	3939 (F)	4335 (F)	4730 (F)	5519 (F)	7893 (F)
21.	9916	6788	15 (T)	87 (T)	159 (T)	304 (T)	739 (T)
22.	9914	3395	9 (T)	48 (T)	96 (F)	214 (F)	739 (F)
23.	8	7	13 (T)	66 (T)	120 (T)	227 (T)	550 (T)
24.	8	5	8 (F)	37 (F)	73 (F)	160 (F)	550 (F)
25.	10	35	24 (T)	68 (T)	113 (T)	202 (T)	472 (T)
26.	0	3	10 (F)	34 (F)	59 (F)	109 (F)	257 (F)
27.	2036	10143	1 (F)	80 (F)	187 (T)	490 (T)	2095 (T)
28.	0	446	12 (F)	61 (F)	120 (F)	264 (F)	909 (F)
29.	22	229	58 (F)	86 (F)	112 (F)	166 (F)	328 (F)
30.	11	156	4 (F)	25 (F)	52 (F)	121 (F)	459 (F)
ABS	0	34410	41939	41935	41923	41941	41946
RMS	0	12694	14254	14195	14144	14323	14355

Table 5: Results for metric K2 (Unlimited Space).

Query	Actual	BestEst	CurrEst	Metric2.1	Metric2.2	Metric2.3
01.	381939	66348	33174 (T)	33832 (T)	33629 (T)	33374 (T)
02.	46284	28356	16169 (T)	16398 (T)	16329 (T)	16237 (T)
03.	46	15171	16587 (T)	17402 (T)	17153 (T)	16833 (T)
04.	5402	4295	4295 (T)	4295 (T)	4295 (T)	4295 (T)
05.	0	105	33174 (F)	33838 (F)	33636 (F)	33374 (F)
06.	6067	6569	15557 (T)	15750 (T)	15691 (T)	15616 (T)
07.	506	4472	4472 (T)	4557 (T)	4531 (T)	4498 (T)
08.	13844	4720	5770 (T)	5787 (T)	5780 (T)	5777 (T)
09.	6792	3468	3468 (T)	3468 (T)	3468 (T)	3468 (T)
10.	8182	8619	8619 (T)	8619 (T)	8619 (T)	8619 (T)
11.	103978	24994	3510 (F)	3579 (F)	3556 (F)	3530 (F)
12.	95979	23640	3317 (F)	3383 (F)	3360 (F)	3337 (F)
13.	83184	21625	3134 (F)	3196 (F)	3177 (F)	3154 (F)
14.	63099	15528	2915 (F)	2974 (F)	2954 (F)	2931 (F)
15.	25177	27	1152 (T)	1175 (T)	1168 (T)	1158 (T)
16.	51030	3016	530 (T)	540 (T)	537 (T)	534 (T)
17.	59039	4380	730 (F)	743 (F)	740 (F)	733 (F)
18.	71819	6487	985 (F)	1005 (F)	998 (F)	992 (F)
19.	91908	13133	1636 (F)	1669 (F)	1659 (F)	1646 (F)
20.	129826	27859	3939 (F)	4017 (F)	3994 (F)	3962 (F)
21.	9916	6788	15 (T)	112 (T)	82 (T)	44 (T)
22.	9914	3395	9 (T)	58 (T)	42 (T)	23 (T)
23.	8	7	13 (T)	83 (T)	61 (T)	34 (T)
24.	8	5	8 (F)	43 (F)	32 (F)	18 (F)
25.	10	35	24 (T)	72 (T)	57 (T)	38 (T)
26.	0	3	10 (F)	39 (F)	30 (F)	19 (F)
27.	2036	10143	1 (F)	162 (T)	110 (F)	46 (F)
28.	0	446	12 (F)	71 (F)	52 (F)	30 (F)
29.	22	229	58 (F)	76 (F)	71 (F)	63 (F)
30.	11	156	4 (F)	30 (F)	21 (F)	11 (F)
ABS	0	34410	41946	41941	41943	41945
RMS	0	12694	14509	14489	14495	14503

Table 6: Table with histogram statistics

Query	Triples	Distinct Subjects	Distinct Objects	Maximum Triples per Subject	Maximum Triples per Object
P1	289753	188568	2336	20	13258
P2	34730	34730	2	35	1888
P3	134113	132097	18676	1	385

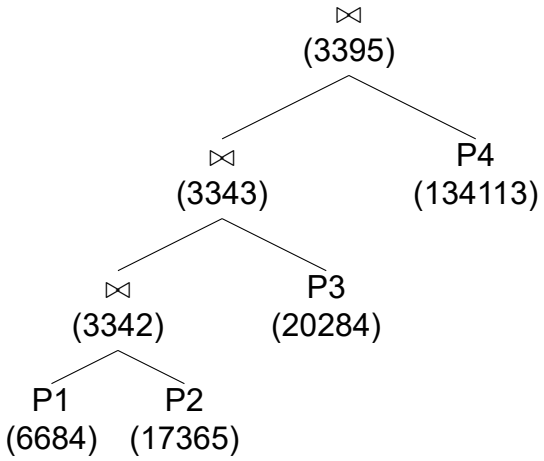

Figure 21: Produced Plan (BestEst).

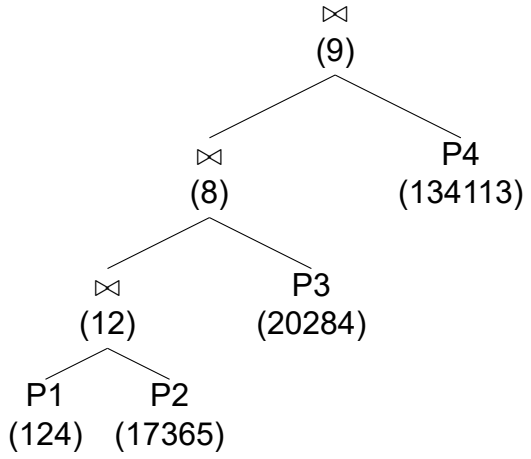

Figure 22: Produced Plan (CurrEst).

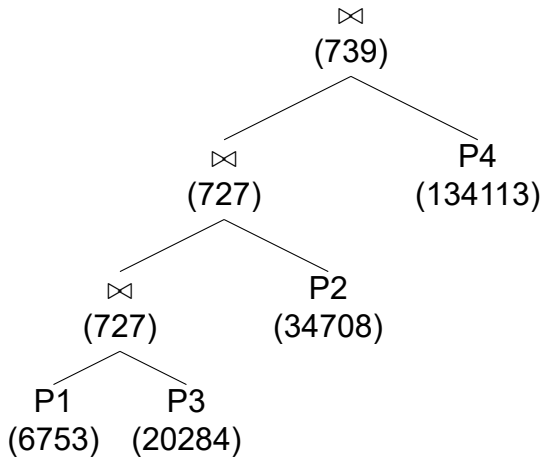

Figure 23: Produced Plan (Metric1.5).

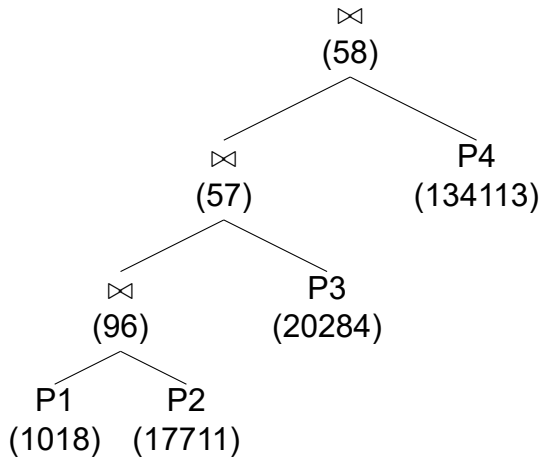

Figure 24: Produced Plan (Metric 2.1).

The estimation that histogram would make without our method (Figure 22) is orders of magnitude smaller but the plan remains the same.

Even when using the metric 1.5 which is the largest overestimation that we used in our experiments we can see that the results still remains very inaccurate. But in this case (Figure 23) (also at 1.3 and 1.4) the plan changes compared to the one with the accurate statistics because P2's estimated cardinality becomes larger than P3's and optimizer prefers P3 to be joined first with P1. So in this case our method has negative impact to the optimizer's choice.

But when metric 2.1,2.2,2.3 is applied (FIgure 24) we notice that the plan remains the same with the first one (BestEst).

So in this experiment we saw a case where our method when metric K1 is used and especially metrics 1.3, 1.4, 1.5 had negative impact to optimizer's plan. Metric K2 because of the modest overestimation that makes did not affect the plan in a negative way. Also this

is a case where we can notice that the independence assumption can lead to pure estimations and bad execution plans. Using our method at these cases we take estimations closer to the actual results but we do not solve this problem.

5.3.2 Insufficient Learning and Limited Space

At tables 7 and 8 we can see the results we got after running the workload. These tables include the actual cardinalities at the Actual column,the cardinality estimated when histogram is provided with the actual statistics for every subquery at the BestEst column and the cardinalities estimated when our method is used.

As we can see both the average ABS error and the average RMS error are very close if we compare our method's results with the previous estimations results. Excluding the Metric1.5 that leads to much larger average error and Metric2.1 that leads to a little smaller average error all the other cases lead to a little larger average error compared to the previous estimations.

5.3.2.1 Results Analysis

Queries where we achieve better performance

QUERY 27

We analyzed this case in detail in section 3.

Queries where we achieve worse performance

In our limited experiments we did not find a case where our method led to a worse plan than the one with the previous estimations. Although we can notice that in most cases the plan chosen either using our method or not is different from the one selected when optimizer was provided with the accurate statistics.

5.4 Conclusion

So in this chapter we used some queries that could be seen in real case scenarios in order to evaluate our method. The evaluation of our method was based in two factors,the accuracy of the cardinality estimation and the impact of our method in optimizer's decision. About the first factor we can conclude that we did not notice a significant change to the histogram's accuracy compared to the histogram's estimation without using our method.Especially when metric K2 or metric K1 with a small value assigned to the constant that make more modest overestimations the average error was really close to the previous estimations.On the other side metric K1 when a small value is assigned to the constant led to bigger average error values. About the second factor we can conclude that we noticed one case where our method changed optimizer's decision that result to a plan which had much less cost than the plan that was selected with the previous histogram's estimations,which was exactly what we wanted to achieve. There was also a case that our method especially when metric K1 was used and constant was assigned with big values had a negative impact to the optimizer's plan.

Table 7: Results for metric K1 (Limited Space).

Query	Actual	BestEst	CurrEst	Metric1.1	Metric1.2	Metric1.3	Metric1.4
01.	381939	66348	60971 (T)	64288 (T)	67608 (T)	74241 (T)	94151 (T)
02.	46284	28356	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
03.	46	15171	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
04.	5402	4295	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
05.	0	105	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
06.	6067	6569	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
07.	506	4472	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
08.	13844	4720	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
09.	6792	3468	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
10.	8182	8619	60971 (F)	64288 (F)	67608 (F)	74241 (F)	94151 (F)
11.	103978	24994	259 (F)	272 (F)	285 (F)	315 (F)	403 (F)
12.	95979	23640	30 (F)	33 (F)	36 (F)	40 (F)	53 (F)
13.	83184	21625	10438 (F)	11011 (F)	11576 (F)	12721 (F)	16149 (F)
14.	63099	15528	5470 (F)	5770 (F)	6071 (F)	6670 (F)	8482 (F)
15.	25177	27	504 (F)	534 (F)	563 (F)	625 (F)	808 (F)
16.	51030	3016	1044 (F)	1106 (F)	1162 (F)	1276 (F)	1623 (F)
17.	59039	4380	1273 (F)	1342 (F)	1414 (F)	1551 (F)	1970 (F)
18.	71819	6487	1273 (F)	1342 (F)	1414 (F)	1551 (F)	1970 (F)
19.	91908	13133	1273 (F)	1342 (F)	1414 (F)	1551 (F)	1970 (F)
20.	129826	27859	1273 (T)	1342 (T)	1414 (T)	1551 (T)	1970 (T)
21.	9916	6788	15 (F)	87 (F)	159 (F)	304 (F)	739 (F)
22.	9914	3395	14 (F)	85 (F)	162 (F)	341 (F)	1049 (F)
23.	8	7	13 (T)	66 (T)	120 (T)	227 (T)	550 (T)
24.	8	5	13 (F)	64 (F)	123 (F)	254 (F)	781 (F)
25.	10	35	12 (F)	1010 (F)	2008 (F)	4004 (F)	9991 (F)
26.	0	3	27 (T)	77 (T)	125 (T)	224 (T)	520 (T)
27.	2036	10143	1 (F)	53 (F)	203 (F)	798 (T)	4939 (T)
28.	0	446	12 (F)	993 (F)	2077 (F)	4549 (F)	14393 (F)
29.	22	229	12 (T)	1025 (T)	2038 (T)	4065 (T)	10143 (T)
30.	11	156	13 (F)	83 (F)	159 (F)	335 (F)	1032 (F)
ABS	0	34410	52835	53747	54661	56488	61976
RMS	0	12694	14690	14720	14759	14862	15368

Table 8: Results for metric K2 (Limited Space).

Query	Actual	BestEst	CurrEst	Metric2.1	Metric2.2	Metric2.3
01.	381939	66348	60971 (T)	61053 (T)	61027 (T)	60997 (T)
02.	46284	28356	60971 (F)	61053 (F)	61027 (F)	60997 (F)
03.	46	15171	60971 (F)	61053 (F)	61027 (F)	60997 (F)
04.	5402	4295	60971 (F)	61053 (F)	61027 (F)	60997 (F)
05.	0	105	60971 (F)	61053 (F)	61027 (F)	60997 (F)
06.	6067	6569	60971 (F)	61053 (F)	61027 (F)	60997 (F)
07.	506	4472	60971 (F)	61053 (F)	61027 (F)	60997 (F)
08.	13844	4720	60971 (F)	61053 (F)	61027 (F)	60997 (F)
09.	6792	3468	60971 (F)	61053 (F)	61027 (F)	60997 (F)
10.	8182	8619	60971 (F)	61053 (F)	61027 (F)	60997 (F)
11.	103978	24994	259 (F)	259 (F)	259 (F)	259 (F)
12.	95979	23640	30 (F)	30 (F)	30 (F)	30 (F)
13.	83184	21625	10438 (F)	10448 (F)	10445 (F)	10441 (F)
14.	63099	15528	5470 (F)	5473 (F)	5476 (F)	5473 (F)
15.	25177	27	504 (F)	504 (F)	504 (F)	504 (F)
16.	51030	3016	1044 (F)	1047 (F)	1047 (F)	1044 (F)
17.	59039	4380	1273 (F)	1276 (F)	1276 (F)	1273 (F)
18.	71819	6487	1273 (F)	1276 (F)	1276 (F)	1273 (F)
19.	91908	13133	1273 (F)	1276 (F)	1276 (F)	1273 (F)
20.	129826	27859	1273 (F)	1276 (F)	1276 (F)	1273 (F)
21.	9916	6788	15 (T)	112 (T)	82 (T)	44 (T)
22.	9914	3395	14 (F)	104 (F)	76 (F)	41 (F)
23.	8	7	13 (T)	83 (T)	61 (T)	34 (T)
24.	8	5	13 (F)	77 (F)	57 (F)	32 (F)
25.	10	35	12 (F)	2175 (F)	1511 (F)	663 (F)
26.	0	3	27 (T)	79 (T)	63 (T)	42 (T)
27.	2036	10143	1 (F)	179 (T)	87 (F)	18 (F)
28.	0	446	12 (F)	2032 (F)	1411 (F)	619 (F)
29.	22	229	12 (T)	2208 (T)	1534 (T)	673 (T)
30.	11	156	13 (F)	102 (F)	74 (F)	40 (F)
ABS	0	34410	52835	53004	52952	52886
RMS	0	12694	14690	14687	14687	14689

6. CONCLUSION

6.1 Summary

In this Thesis we developed a method in order to avoid some really bad optimizer's choices that would lead to execution plans with very large costs. Especially in the field of semantic web where there are other significant sources of latency such as the communication between the distributed components and both the size of data and the distributed sources that a query may involve can be enormous,such a choice can lead to a latency that is not accepted by the end users. we noticed that these cases are rare,this is the reason why adaptive histograms can not adapt to them, but when a query falls to these cases the estimations are very inaccurate. Another thing that we have noticed was that in the cases of queries with multiple joins,that in the field of semantic web are very common, histogram's estimations were at most cases a lot smaller than the actual.

Therefore we invented a method that was taking into account the biggest outliers in order to make the estimation and we forced the histogram to make overestimations compared to the estimations that was making before. This may had a negative impact to the average case.But our overestimation in the average case had not significant cost either in the cardinality estimation or the execution plan. On the other hand there were some rare cases where a big outlier was involved that our method made a significant difference compared to both the previous histogram's cardinality estimation and the optimizer's execution plan choice.

The definition of our method also includes some parameters that can be assigned by the user. We can conclude that metric K2 has good performance in the average case and can also make an improvement to the optimizer's plan even though the overestimation that it makes is more modest than the metric K1.The estimation had no big difference when different values were assigned to the parameter but in our experiments the smallest value that is equal to 2 had the better performance. On the other hand metric K1 when big values are assigned to its parameter can lead to really big overestimations that result in very inaccurate estimations. In our experiments metric K1 had the better performance when 10 was assigned to the parameter but also values 5 and 20 had also similar performance.

Although we achieved to have a better performance at cases where big outliers are involved there are cases where our histogram still makes poor estimations. One of them that we referred to at the experiments chapter is the independence assumption that we make when we estimate the cardinality of a join. Also as seen from the experiments our method does not have good performance when a lot of merges are taking place during histogram construction.

6.2 Future Work

Future work could include a method to keep statistics of more complicated queries such as join queries and not only statement patterns. It is impossible to keep statistics of all the join queries due to limitation of space but there could be invented a new method or there could be used one that exists in bibliography in order to make a selection of the statistics that are more significant.

ABBREVIATIONS - ACRONYMS

DBMS	Database Management System
ABS	Absolute Estimation Error
RMS	Root Mean Square Error
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema

APPENDIX A. THE WORKLOAD

```
01. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/Artist> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
02. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/SportsTeam> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
03. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/ComicsCharacter> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
04. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/Language> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
05. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/FootballTeam> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
06. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/Airline> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
07. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/College> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
08. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/BasketballPlayer> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
09. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/Boxer> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
10. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type <http://dbpedia.org/ontology/Weapon> .
        ?s foaf:name ?name.
        ?s skos:subject ?sub.
}
```

```
11. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.     ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate >= '1970-01-01'^^xsd:date)
}
12. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate >= '1980-01-01'^^xsd:date)
}
13. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate >= '1990-01-01'^^xsd:date)
}
14. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate >= '2000-01-01'^^xsd:date)
}
15. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate >= '2010-01-01'^^xsd:date)
}
16. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
 FILTER (?releaseDate <= '1970-01-01'^^xsd:date)
 }
17. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate <= '1980-01-01'^^xsd:date)
}
18. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film  .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
```

```
FILTER (?releaseDate <= '1990-01-01'^^xsd:date)
}
19. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate <= '2000-01-01'^^xsd:date)
}
20. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film .
        ?s foaf:name ?name.
        ?s skos:subject ?subject.
        ?s dbo:releaseDate ?releaseDate.
FILTER (?releaseDate <= '2010-01-01'^^xsd:date)
}
21. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject <http://dbpedia.org/resource/Category:American_films>.
        ?s foaf:name ?title.
        ?s dbo:releaseDate ?o.
}
22. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film.
        ?s skos:subject <http://dbpedia.org/resource/Category:American_films>.
        ?s foaf:name ?title.
        ?s dbo:releaseDate ?o. }
23. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject
              <http://dbpedia.org/resource/Category:Films_directed_by_Ted_Demme>.
        ?s foaf:name ?title.
        ?s dbo:releaseDate ?o. }
24. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s  rdf:type dbo:Film.
        ?s skos:subject
              <http://dbpedia.org/resource/Category:Films_directed_by_Ted_Demme>.
        ?s foaf:name ?title.
        ?s dbo:releaseDate ?o. }
25. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject
              <http://dbpedia.org/resource/Category:German-language_writers>.
        ?s foaf:name ?name.
        ?s rdfs:label ?o.
}
26. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject
              <http://dbpedia.org/resource/Category:Novels_based_on_the_Bible>.
        ?s foaf:name ?title.
        ?s dbo:releaseDate ?o.
}
27. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject <http://dbpedia.org/resource/Category:Living_people>.
        ?s foaf:name ?title.
        ?s rdf:type dbo:Artist .
```

```
}
28. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject <http://dbpedia.org/resource/Category:Greek_mythology>.
        ?s foaf:name ?title.
        ?s rdf:type dbo:Film.
        ?s dbo:releaseDate ?o.
}
29. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject
             <http://dbpedia.org/resource/Category:BBC_radio_comedy_programmes>.
        ?s foaf:name ?title.
        ?s dbo:releaseDate ?o.
}
30. SELECT * FROM <http://dbpedia3.2.org> WHERE {
        ?s skos:subject <http://dbpedia.org/resource/Category:American_actors>.
        ?s foaf:name ?title.
        ?s  rdf:type <http://dbpedia.org/ontology/Comedian> .
}
```

REFERENCES

[1] Christos Anagnostopoulos and Peter Triantafillou. Query-driven learning for predictive analytics of data subspace cardinality. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):47, 2017.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[3] Dan Brickley. Resource Description Framework (RDF) schema specification 1.0, W3C candidate recommendation. *http://www. w3. org/TR/2000/CR-rdf-schema-20000327*, 2000.

[4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: a multidimensional workload-aware histogram. In *ACM SIGMOD Record*, volume 30, pages 211–222. ACM, 2001.

[5] Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. SemaGrow: Optimizing federated SPARQL queries. In *Proceedings of the 11th International Conference on Semantic Systems*, pages 121–128. ACM, 2015.

[6] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[7] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.

[8] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[9] Andranik Khachatryan, Emmanuel Müller, Christian Stier, and Klemens Böhm. Improving accuracy and robustness of self-tuning histograms by subspace clustering. *IEEE Transactions on Knowledge and Data Engineering*, 27(9):2377–2389, 2015.

[10] Ora Lassila and Ralph R Swick. Resource description framework (RDF) model and syntax specification. 1999.

[11] Guy M Lohman. Is query optimization a “solved” problem. In *Proc. Workshop on Database Query Optimization*, page 13. Oregon Graduate Center Comp. Sci. Tech. Rep, 2014.

[12] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993, 2009.

[13] Eric Prud, Andy Seaborne, et al. SPARQL query language for RDF. 2006.

[14] Utkarsh Srivastava, Peter J Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. ISOMER: Consistent histogram construction using query feedback. In *Data Engineering, 2006. ICDE’06. Proceedings of the 22nd International Conference on*, pages 39–39. IEEE, 2006.

[15] W. E. Winkler. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. Technical report, Proceedings of the Section on Survey Research Methods, American Statistical Association, pp. 354–359, 1990.

[16] Katerina Zamani, Angelos Charalambidis, Stasinos Konstantopoulos, Nickolas Zoulis, and Effrosyni Mavroudi. Workload-Aware Self-tuning Histograms for the Semantic Web. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII*, pages 133–156. Springer, 2016.