Master Thesis

Online Datalog on RDF Knowledge Graphs: Top-down or Bottom-up?

Author:
Varsha Ravichandra Mouli
Student number:
2593036

Supervisor:
Dr. Jacopo Urbani

5th July, 2019

Contents

1 Introduction **5**

2 Background **11**
2.1 RDF 11
2.2 SPARQL 12
2.3 Reasoning 13
2.4 Datalog 14
2.5 QSQ 16
2.6 Magic Sets 16
2.7 VLog 17

3 Reasoning Complexity Features **19**
3.1 Algorithm based data structures 19
3.2 Feature selection 21

4 Algorithm for Estimation Features **25**

5 Top Down or Bottom up **29**
5.1 Query Generation 31
5.1.1 Manual query creation 31
5.1.2 Query Generation 33
5.1.3 Query Generation from rules file alone 34

6 Evaluation **37**
6.1 Experiment Settings 37
6.2 Results 38
6.2.1 Training Queries generation 39
6.2.2 Feature Generation 40

6.2.3 Threshold Identification 41
6.2.4 Evaluation of prediction model 44

7 Related Work **53**

8 Future works and conclusions **55**

Appendix **56**

A **57**
A.1 Naming Convention 57
A.2 Scatter plots 57

VRIJE UNIVERSITEIT AMSTERDAM

Abstract

Faculty of Sciences
Department of Computer Science

Online Datalog on RDF Knowledge Graphs:Top-down or Bottom-up?

by Varsha Ravichandra Mouli

The Semantic Web community has studied extensively the task of materializing efficiently all possible derivations for a KG, typically an offline task. We focus on the problem of efficiently performing online reasoning, i.e., computing efficiently all relevant implicit answers during query answering. We focus on reasoning that can be encoded with Datalog programs. Based on the implementation of two methods for query-driven Datalog evaluation: Query-Subquery(QSQ), a top-down procedure, and Magic Set (MS), another method which proceeds bottom-up, we know one implementation will be preferable to other depending on the amount of reasoning triggered by the query.
Thus, to achieve the highest efficiency, we study how we can estimate whether a given query will be faster with QSQ or MS. We propose a number of heuristics for making such estimates, and identify thresholds for each of them and model the decision making as a binary classification problem for each individual feature. Our experiments conducted on various KGs and using the VLog engine, showed that our estimators were able to choose the right algorithm in most of the cases. This increases significantly the efficiency of reasoning at query-time.

Chapter 1

Introduction

The Web contains in it collections of interlinked information which are aimed to cater to human consumption. In other words, Web contains data in human readable documents, that are interlinked with each other making them navigation-able. Semantic Web[4] though being an extension of Web, conceptually has it differences. It consists of a collection of data that can be processed by machines and are hence in machine readable format.

This leads us to the different types of data that reside on the Web. Some of it are structured and others unstructured. Structured data have a pre-defined model and are referred and identified by meta-data tags. These data can be stored in definite fields and are machine readable. Unstructured data do not have any defined model and are often text files. These structured and detailed information available over the Web are often represented by Knowledge Graphs (KG).

The information available in a KG can be presented in Resource Description Framework(RDF) statements [11]. They consist of RDF triples whose terms can be URIs, blank nodes, or literal values. The triples have the format of: a *subject*, a *predicate*, and an *object*. RDF formatted data is useful as it provides a large repository of serialized knowledge, in a directed and labelled graph form. This allows for performing tasks such as inferring implicit and previously unknown information from the KG. We refer to this process as reasoning. RDF data has application especially in academic research and the web.

Hence, with the extensive amount of information available, it is crucial in many cases to consume all knowledge contained in them, via rea-

soning. In order to perform reasoning, we need systems that can work on the knowledge graph by querying them for explicitly available information, and also all the content that needs to be derived from them. There is existing research on reasoning applications[16] [6].
Working with an example to understand the concept of reasoning, given a program P :

$$Employee(x) \leftarrow Person(x) \wedge Employer(x, y) \wedge Company(y)$$
$$Person(x) \leftarrow Employee(x)$$
$$Person(x) \leftarrow Employer(x, y)$$
$$Company(y) \leftarrow Employer(x, y)$$

and a set of known facts :

$$Employer(Alice, ABC) Employer(Bob, XYZ)$$

The derived facts can be :

$$Person(Alice), Person(Bob)$$
$$Company(ABC), Company(XYZ)$$
$$Employee(Alice), Employee(Bob)$$

Reasoning can be a simple process if the data available to process is considerably small. But, since most of the available KGs contain billions to trillions of tuples [22], reasoning can face challenges such as :

- Time constraints
- Correctness of results
- Relevancy
- Resource availability

The process of reasoning can be performed earlier(via materialization) or during query time(backward reasoning). The former can be described as an offline task and the latter as online.

The current trend in reasoning revolves around materialization. Materialization is a form of pre-computation which is done to obtain all possible derivations from a given input explicitly. This happens before the user interacts with the KG and a substantial amount of time is consumed during this process. This guarantees that no reasoning is needed when the user queries the knowledge base. Query evaluation can thus proceed extremely quickly due to the pre-processing done .

But, materialization is a computationally resource and memory intensive task, especially if the given input is large. It can take from a few hours to days for the process to complete. Research by [10] shows for LUBM8000 with 1.1 billion tuples, it takes 11 hours for materialization to complete. To reduce the time consumed, research has been done on Webpie[21] and Dynamite[19] which exploits the concept of parallelism thus improving the performance, but, this means memory and processor requirement become substantial. There is further research done on parallel processing and main-memory computation[12] and also optimization's to avoid redundant inferencing[17] allowing to improve resource usage and reduce time usage.

As a part of this thesis, we focus on reasoning performed in real time. A user would query the KG in an online fashion which means due to lack of materialization, reasoning can be complex and can consume time. But it is essential that the reasoner does not take a long time due to the real time characteristic. It is thus essential that time is not wasted in computing intermediate derivations to provide answers. Relevancy is also a key factor as the reasoner should compute necessary derivations rather than retrieving all available derivations. Reasoning in an online fashion allows us to overcome the consequences which arise whenever input data changes, which is usually unacceptable for materialization.

Thus, for online reasoning, Datalog based query-driven algorithms like Query-Subquery (QSQ) and Magic-set rewriting (MS) prove to be ideal[1].

QSQ performs reasoning in a method that follows a top-down approach. It uses the SLD framework for the focus to be contained on relevant facts and thus the production of tuples that do not contribute to the answer tuple can be avoided. The main idea being that the given input query is split into a number of sub-queries. The results produced by the sub-queries can be further used by the reasoner to calculate all implicit answers. All this follows a top-down evaluation of the rules, starting from the input query.

Magic sets, on the other hand performs reasoning in a method that follows a bottom-up approach. The input set of rules are pre-processed before the input query is evaluated on them. The magic set approach transforms the input query into a new query, which computes the same answer as that of the original query.

Working on the current implementation of QSQ and MS on VLog, it

is observed that working of both the algorithms are theoretically similar and provide the same results. But for certain queries one algorithm performs faster than the other. Few of the reasons that can be attributed to this are : QSQ allows selections from the input query to be propagated through the rules as they are expanded. This information cannot be directly utilized in a bottom-up approach. Another reason is that QSQ works on the original program without any rewriting , while MS requires an initial rewriting and relies on a materialization engine for the bottom-up execution of the rules. The rewriting allows better performance when in the case of extensive reasoning. Hence, it becomes essential that we are able to identify whether the startup cost associated with rewriting the query trumps the advantages of rewriting?

If more reasoning can imply MS, then it would be easy to suggest that input queries with no complex reasoning can use QSQ for reasoning. But, we would require an estimation strategy, which for any input query, would be able to fetch complexity of reasoning. This would help in classifying for a given query, which algorithm will perform better: a top-down procedure (QSQ) or a bottom-up one (MS).

But with the current state of implementation of both the algorithms, it would not be feasible to classify that queries that require substantial reasoning might perform better with MS than with QSQ. Thus, for a given input query, the question of which is the faster algorithm is a difficult one to answer.

Based on the above discussions, we have formed the research question we try to answer via the thesis :

> Considering the process of Online Reasoning on a KG, is it possible to determine an estimation based strategy, which when given an input query, will be able to determine which algorithm performs better? QSQ or MS.

To answer the research question raised above, we have identified a solution. We perform an analysis on both the algorithms QSQ/MS to identify estimation features that contribute to the complexity and execution time. We propose an algorithm that runs on QSQ to a certain depth to fetch values for these features. We then generate training queries for a given KG and run the estimation procedure on the queries. Based on the features obtained, we graph a scatter plot that

helps in identifying relationship between features and decision. We fix the threshold for each feature based on the relationship and model this as a rule for a binary classification in VLog.

We work with the rule language, Datalog [1] that is used to query and perform reasoning on the Knowledge graph to compute all derivations. They consist of *if-then* rules which can be recursive in nature.

We have currently tested our approach on LUBM(1000) [8] and DBPedia[3] and identified that we are able to correctly predict the algorithm for a substantial amount of queries.

This thesis consists of 8 chapters that is largely based on a scientific paper under submission.
This thesis document is structured as follows: In chapter 2, we will provide some background on SPARQL , Datalog, Reasoning and on algorithms QSQ and MS. In chapter 3, we will discuss our approach in analyzing QSQ and MS to identify estimation features and in chapter 4 discuss the implementation of the algorithms for estimation. Chapter 5 will discuss query generation leading to chapter 6 where we discuss benchmark results showing how well our algorithms perform. Chapter 7 will discuss the related works associated with the thesis and in chapter 8 we provide the conclusion and possible future works.

Chapter 2

Background

In this chapter we focus on describing the technologies that are used in the thesis. This will help in understanding the rest of the document.

Section 2.1 contains a brief description of RDF while section 2.2 does the same for the SPARQL language. In Section 2.3 we discuss process of reasoning, section 2.4 provides information on how Datalog is defined while section 2.5 and 2.6 describe the working of two reasoning algorithms, QSQ and Magic sets. Section 2.6 introduces VLog.

2.1 RDF

Information on the web available over KG are often represented with the RDF data model. RDF terms are represented as triples. The triples have the format of : a *subject*, a *predicate*, and an *object*. These can be URIs, literals or blank nodes. URI's are Uniform Resource Identifiers used to denote resources, blank nodes denote anonymous resources and literals represent values. Given any RDF statement, subject is always either URI or blank nodes, object can be URI or literal or blank nodes and predicate can hold either URI or blank nodes.

An example that would help in understanding is "Person1 likes computers", where Person1 is the subject, likes is the predicate and computers is the object.

RDF triples can be represented in many ways:

- JSON-LD [1]
- N-Triples [2]
- N-Quads [3]
- Notation3 [4]
- RDF/XML[5]
- Turtle [6]

To define the above concept , an example of a RDF triple represented in N-Triples format can be given by :

```
<http://www.cs.vu.nl/>
<lubm:subOrganizationOf>
<https://www.vu.nl/>
```

The above RDF statement shows that the concept identified by Uniform Resource Identifier(URI) `<http://www.cs.vu.nl/>` is a subOrganizationOf another concept, identified by the URI `<https://www.vu.nl/>`.

2.2 SPARQL

SPARQL[7] is a query language for knowledge graphs, that is capable of working and fetching RDF data.

SPARQL is SQL like in nature where the query is structured as a RDF triple. The variation from RDF is that any of the subject, predicate or object can be represented as a variable.

An example of a simple SPARQL query to show the list of all the members belonging to Department0 of Univeristy0 is:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl#>
```

[1] https://www.w3.org/ns/json-ld
[2] https://www.w3.org/TR/n-triples/
[3] https://www.w3.org/TR/n-quads/
[4] https://www.w3.org/TeamSubmission/n3/
[5] https://www.w3.org/TR/rdf-syntax-grammar/
[6] https://www.w3.org/TR/turtle/
[7] https://www.w3.org/TR/rdf-sparql-query/

```
SELECT ?x
WHERE {
?x lubm:memberOf <http://www.Department0.University0.edu>.
}
```

where
URI for the object is <http://www.Department0.University0.edu/,
URI for the predicate is <http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl#memberOf>.

The sample answer to the query, represented in N-Triples format is :

```
<http://www.Department0.University0.edu/AssociateProfessor13>
<http://www.Department0.University0.edu/FullProfessor4>
<http://www.Department0.University0.edu/Lecturer5>
<http://www.Department0.University0.edu/GraduateStudent121>
<http://www.Department0.University0.edu/UndergraduateStudent104>
```

2.3 Reasoning

Reasoning can be defined as the process of applying existing knowledge on new situations and inferring further knowledge from them.

With respect to this thesis, to better understand reasoning we can work with an example:

```
http://www.FullProfessor0.edu
ub:worksFor
http://www.Department0.edu

http://www.Department0.edu
ub:subOrganizationOf
http://www.University0.edu
```

The first RDF triple states that *FullProfessor0 works for Department0* and the second RDF triple states that *Department0 is a sub organiza-*

tion of University0. Using the above information, we can deduce that

```
http://www.FullProfessor0.edu
ub:worksFor
http://www.University0.edu
```

In the above example, we considered all available information that was true and performed reasoning to derive all possible deductions. It is time consuming to derive all facts. Another approach would be to consider one single statement that needs to be verified and move step-by-step via the original information till every input fact is proved to be true.

In our thesis, we focus on the second approach which is called Backward Chaining.

2.4 Datalog

Datalog is a widely used rule based query language which can work with recursive rules. It has many applications [9] but we focus here on its ability to perform query based answering and also reasoning on Knowledge graphs. As a part of this thesis, we work with the usual representation and notations used in Datalog literature. We briefly summarize the basic concepts of datalog that is used in this thesis.

We define a set **C** of *constants*, a set **P** of *predicates* and a set of **V** of *variables*. A *term* is a variable $x \in V$ (x,y,z,etc.) or a constant $c \in C$ (a,b,A,B,etc.). An *atom* is an expression of $R(A_1, ..., A_n)$ consisting of R as a relation/predicate and $A_1...A_n$ as terms. A *fact* is a variable free atom. A *database instance* is a finite set *I* of facts.

In Datalog, the answers of queries are computed using rules where a *rule r* is an expression represented in the form :

$$H \leftarrow B_1, ..., B_n$$

where H and $B_1, ..., B_n$ are atoms. The first atom , H, present on the left side is the *head* atom and the remaining $B_1, ..., B_n$ are *body* atoms.
We use *head(r)* and *body(r)* to refer to the head and body of *r* respectively. A *program* is a finite set P of rules. It is mandatory by enforcement in Datalog that every variable present in the head should atleast

be present once in the body. A *query* Q, is a rule without a head where the atoms can contain variables or constants.

An *extensional predicate (EDB)* occurs only in the body of the rules. An *intensional predicate(IDB)* is a relation occurring in the head of some rule of P.

For every *atom*, an *adornment* can be identified. Every term in the atom $R(A_1, ..., A_n)$ can either be free (*f*) or bound (*b*), i.e., : $A_1, ..., A_n \in (f, b)$ during a query driven computation. It is denoted by the expression R^{bf} , where the adornment 'bf ' appears in the superscript. Given a rule r, it can be rewritten to a form where all atoms are adorned to emphasize the adornments. This is called an *adorned rule*.

A substitution is a partial mapping of variables to terms. Lets denote the same by σ. Given a database D and program P, our objective is to calculate the set of atoms that we can infer using the rules. Let $r(D) = \{\alpha\sigma | \beta_1\sigma, \ldots, \beta_n\sigma \in D\}$ and $P(D) = \bigcup_{r \in P} r(D)$ be the set of facts derived by the rule r and all the rules in P respectively. Further, we set $P^0(D) = D$ and recursively define $P^{i+1} = P(P^i(D))$ for $i \geq 0$. The *materialization* of D with P is the union $P^\infty(D) = \bigcup_{i \geq 0} P^i(D)$. This union is finite due to the safeness of the rules and the finite number of constants. The materialization is computed using a technique called seminaive evaluation (SNE). For all $i > 1$, this technique redefines $r(P^i(D))$ as $r(P^i(D)) = \{\alpha\sigma \mid \beta_1\sigma, \ldots, \beta_n\sigma \in P^{i-1}(D) \wedge \exists \beta_j\sigma \in P^{i-1}(D) \setminus P^{i-2}(D)\}$ to avoid the derivation of duplicates produced in previous stages of the computation.

For our purposes, we are interested on computing only the subset $D_q \subseteq P^\infty(D)$ that is relevant to answer an input query. An input query consists of a single atom. Let $\gamma = p(\mathbf{t})$ be an example of such a query. Our goal is to compute the set of substitutions $Ans(\gamma) = \{\sigma \mid \gamma\sigma \in P^\infty(D)\}$ as quickly as possible, avoiding whenever possible, the computation of the entire $P^\infty(D)$.

Datalog supports query driven approach for evaluation of rules. A primary motivation is to avoid the production of tuples that are not needed to derive any answer tuples. To this end, two approaches are used : Top-down approach and Bottom-up approach. Top down focuses on direct evaluation , where the query contains bound values which are propagated through the rules helping in restricting the search and retrieving relevant facts. Bottom up approach performs initial pre processing of the rules in the given datalog program.

Two popular algorithms for such query-driven computation are Query-Subquery(QSQ) and Magic sets(MS) which are discussed in the next two sections.

2.5 QSQ

QSQ(Query-Subquery)[1] is a top-down technique for evaluation of datalog programs that follows a direct evaluation process. A selection based query is supplemented with the program, for e.g

Anc('Alice',y).

where 'Alice' is a constant.

QSQ uses the SLD framework for the focus to be contained on relevant facts and thus the production of tuples that do not contribute to the answer tuple can be avoided. We begin with the constants available in the query and subsequently calculate the adornment for the query. For all the rules in the program having the same head predicate as the query, these rule adornments are created.

The top-down evaluation aids in the creation of sub-queries for those rules that contain constants. Two temporary data structures are created : $input*$ and $ans*$. $input*$ stores the sub-queries that are evaluated and $ans*$ the facts that are obtained. The constants are pushed from queries to sub-queries. The concept of sideways information passing is used to propagate this information of bound terms from one atom to another. To aid this, another data structure called Supplementary relations $Suppl_1, ...Suppl_(n-1)$ are created to store intermediate information. The $suppl*$,$input*$ and $ans*$ together form the QSQ template.

2.6 Magic Sets

Magic set(MS)[1] is a bottom-up technique for evaluation of datalog problems that performs initial pre-processing of the rules in the datalog program. Given a datalog query q for a P, the magic set approach transforms it into a new query Q_m P_m where the new program is created based on the query and the new program computes the same output as

that of old one. The magic sets are the values stored in the relations $input*$ and $suppl*$ of the QSQ algorithm. Here too, a selection based query is used. Similar to QSQ, the magic set algorithms begins with creation of adorned datalog rules. It creates the program introducing $input$ and $suppl$ as the new predicates.

2.7 VLog

VLog [18] is a system that performs Datalog based in-memory materialization in a new column oriented approach. VLog is coded in C++ and was designed to work on commodity machines.
This materialization is done over large KGs in an efficient way by reducing memory and CPU power consumption. Traditionally new facts that are generated are stored in a row based manner, VLog stores the inferred facts column-by-column. The columnar-oriented storage is effective because columns can be re-used more easily and in practice this leads to significant memory savings. Moreover, VLog adopts an append-only mode for storing the derivations, introduces novel techniques to filter out duplicates, and can interface with different backends like MySQL and Trident.

Chapter 3

Reasoning Complexity Features

We begin with analyzing QSQ and MS algorithm and its implementation over VLog. This helps in understanding the templates and structures used for data storage and propagation. This led us to identify seven features($Est1$ .. $Est7$) which can be used as indicators of the runtime. These features are identified to be contributing factors for reasoning complexity and play a role in deciding runtime of a query.

Section 3.1 discusses the results of analysis of QSQ and MS that helps in identifying the features and section 3.2 defines the estimation features.

3.1 Algorithm based data structures

QSQ and MS framework uses templates/data structures for storing appropriate information during intermediate stages of an evaluation. These data structures hold values that contribute to the runtime of the algorithms.

Lets work with a datalog program:

$$Anc(\boldsymbol{x,y}) \leftarrow Par(\boldsymbol{x,y}) \tag{3.1}$$

$$Anc(x, y) \leftarrow Par(x, z), Anc(z, y) \tag{3.2}$$

We study the following data structures to identify the appropriate

estimation features.

Supplimentary Relations

Considering the program P defined in Section 3.1, to evaluate the subquery Anc^{bf}, if there is a left-to-right evaluation it will lead to the subquery involving Anc^{fb}. We rewrite the query as :

$$Anc^{bf}(x, y) \leftarrow Par(x, z), Anc^{fb}(z, y)$$

which becomes the adorned rule. The adornments of IDB predicates in rule bodies are used for the evaluation of sub-queries. EDB predicates are omitted. We define a data structure that will remember all values needed during a left-to-right evaluation of a subquery. They are defined as *supplementary relations* or symbolically *suppl**. For a rule, with *n* body atoms, a total of *n+1 suppl** are created.

InputTable

Given the query, to initialize the process, the initial value is loaded into a data structure defined as *InputTable* or symbolically *input**.
Considering our program P with query

$$Anc('Alice', y)$$

*input** is stored with 'Alice' . The rule is rewritten based on the bound variable and the evaluation of a rule occurs whenever there is a rule with a matching head predicate and there are tuples in *input** that have not yet been processed for this rule.

OutputTable

When there are no new tuples in *input** and all have been processed, the *OutputTable* or symbolically *ans** will hold the tuples that were generated by the subqueries based on the query IDB predicate.

3.2 Feature selection

Based on the 3 data structures described in section 3.1, we can summarize the following information.

- Up until *input** contains values, intermediate queries are created and evaluated .
- The *ans** contains facts that are answers obtained when answering a query q. The count of this value may vary based on the input query.
- If there are tuples in a supplementary variable

  $$suppl*_i^j$$

  that are yet to be consumed, joins and projections are done on the tuple and predicate R for identifying the tuples that need to be transferred from one supplementary relation to the next.
- Based on the input query, the number of rules used to derive the output facts vary.

Based on the above facts, we identified 7 different features that contributes to the complexity of query execution and runtime.

- $Est1$: isSubjectBound : This is a boolean feature which has the value 1 if the query's subject is bound (i.e. if subject is constant), or 0 otherwise.

- $Est2$: isObjectBound : This is a boolean feature which has the value 1 if the query's object is bound (i.e. if object is constant), or 0 otherwise.

- $Est3$: numberOfResults : For a given ruleset R and database D, if a query Q is fired, there would be a set of relevant facts that answer Q. When running QSQ as the algorithm the facts for Q are stored in the data structure of $ans*$ and when MS is being run they are stored by the underlying engine that evaluated the rewritten rules. The count of these facts provide the numberOfResults

- $Est4$: costOfComputing : When a query Q is fired on a database D, multiple rules of the form $H \leftarrow B_1, ..., B_n$ where H and $B_1, ..., B_n$

representing atoms are executed. To derive the facts in $ans*$, we must execute the rule r for the form $\alpha \leftarrow \beta_1, \ldots, \beta_n$ to a partially augmented database $P^i(D) \subseteq P^\infty(D)$ and let $I_r = \bigcup_{i=1}^{n}\{\sigma \mid \beta_i\sigma \in P^i(D)\}$. Among the various body atoms in a given rule, there is repeated substitutions till they can be mapped onto the head rule. This is an intensive task, that would contribute to a large extent towards the total execution time. However,there is no linear relationship between the size of I_r and the complexity and time taken to perform join.

- $Est5$: numberOfRules: From the program P, if there is a rule r that has a head predicate matching that of the predicate from query q and there are tuple(s) in the $input*$, evaluation of rule can happen. The tuple that unifies with the head of the rule are populated into the $suppl*$. Rule execution proceeds by cascadingly populating $suppl*$ by passing new tuples or generating tuples based on new IDB predicates.
  If the 0^{th} $Suppl*$ is populated for a rule r, the rule gets executed and we count the total number of rules executed for a input query. Rules can be executed multiple times to derive new facts and all of them are counted. The multiple execution times depends on the depth.

- Est_6 : numberOfQueries : QSQ being a top-down evaluation creates sub-queries for IDB predicates. $input*$ is used to form sub-queries represented as (Anc^γ,InputTable), if *Anc* is the predicate name of the query under consideration. Up until $input*$ contains tuples that have not been processed for a given rule, sub-queries are generated. The number of times $input*$ gets populated with tuples, determines the number of sub-queries created. MS rewrites the program P and $\{p(\mathrm{t}) \mid p(\mathrm{t}) \in P^\infty_\gamma(D) \wedge p$ starts with 'input'$\}$ is the set of sub-queries. The sub-queries are stored to prevent infinite query execution and can contribute to the total execution time.

- Est_7 : numberofUniqueRules : For a given query Q, it is the count of distinct rules that were executed to obtain results. Here, we do

not count any rule twice, even if they derive new facts during the process.

Chapter 4

Algorithm for Estimation Features

Now that we have identified the features that would contribute to the execution time as mentioned in section 3.2, the next step is to fetch values for the same. This process would be non-optimal if an actual evaluation based on both the algorithms takes place. We need to thus determine an estimation procedure that has better execution time when compared to the actual execution time of the evaluation procedure for both the algorithms, but at the same time provides a rough approximation that helps in the decision making process.

The values for isSubjectBound and isObjectBound can be identified from the query itself.

We propose an algorithm which simulates the execution of a top-down evaluation procedure up to a certain depth. The depth can be varied and we can use it to identify if the estimation procedure will improve efficient prediction as the depth increases.

During the execution, this algorithm should keep track of the following :

- The cost of complexity, in terms of join computation, required to execute a rule.
- The number of inferred facts that can potentially be derived.
- Number of sub-queries that are generated.

The algorithm is just an approximation procedure. It provides us information which later is used in rule modelling for identifying QSQ

or MS. It would not be feasible to provide an accurate prediction.

Algorithm 4.1: Routines for estimating metrics like number of queries etc. for a Datalog atom γ on D and program P. $maxDepth$ is a global input parameter.

Data: D = Database, P = Program , γ = Datalog Atom , maxDepth
Result: $answers, cost, queries$
function `estimateRule`$(\sigma, r, D, P, depth)$
  $answers \leftarrow 0, cost \leftarrow 0, queries \leftarrow 0$
  for $\beta_i \in \beta_1, \ldots, \beta_n$ **do**
    $a, c, q \leftarrow$`estimateQuery`$(\beta_i\sigma, D, P, depth)$
    $answers \leftarrow answers + a;$
    $queries \leftarrow queries + q$
    $cost \leftarrow cost * a$
  end
  return $\langle answers, cost, queries \rangle$
function `estimateQuery`$(\gamma, D, P, depth)$
  if $depth \geq maxDepth$ **then**
    return $\langle 1, 1, 0 \rangle$
  end
  $queries \leftarrow 1, answers \leftarrow cost \leftarrow |\{\sigma \mid \gamma\sigma \in D\}|$
  for $\forall r \in P$ **do**
    r is of the form $\alpha \leftarrow \beta_1, \ldots, \beta_n$
    if $\exists\sigma \mid \alpha\sigma = \gamma$ **then**
      $a, c, q \leftarrow$`estimateRule`$(\sigma, r, D, P, depth + 1)$
      $answers \leftarrow answers + a$
      $cost \leftarrow cost + c;$
      $queries \leftarrow queries + q$
    end
  end
  return $\langle answers, cost, queries \rangle$

The estimation algorithm computes the Number of queries, Number of facts, cost of computation and sub-queries generated for a datalog atom γ on D and program P. The pseudocode of the algorithm is described in Algorithm 6.1.

We describe two functions, `estimateQuery` and `estimateRule`. The

`estimateQuery` takes in the input query γ, information from the database D, the program P and depth parameter as inputs. The $maxDepth$ parameter is a global input parameter and the estimation functions are executed to the count of the $maxDepth$ paramater. The depth is initialized to 0 and is incremented during every iteration. This is compared to the $maxDepth$ to identify the termination of iterations as mentioned above. The function returns the estimated metric information for numberOfResults(***Est3***) , costOfComputing(***Est4***) and numberofQueries(***Est6***). The initial function call sets the numberofQueries(***Est6***) to the minimum value of 1 as at least one look up is performed in the database D. Based on the query γ, the finite set of facts that can be retrieved from the database D is populated in the numberOfResults(***Est3***) and costOfComputing(***Est4***). From the program P, all the rules are considered and if there is a rule r that has a head predicate matching that of the predicate from input query γ, or if the tuple is unifiable with the head, then the `estimateRule` function is invoked with the necessary parameters. The counter variable for calculating the numberOfRules(***Est5***) is increased. The call to $estimateRule$ begins with initializing the estimators to 0. For all the body atoms unified with the query, the call is made to $estimateQuery$. The counter variable for calculating the numberOfQueries(***Est6***) is increased. Based on the return values from the call, the numberOfResults(***Est3***) are summed and stored. Every body atom for a given rule calls the $estimateQuery$ and returns a set of answers. The costOfComputing(***Est4***) can be determined as the join cost while performing a cartesian product between all the answers.

To summarize the estimators obtained:

- numberOfQueries(***Est6***) is calculated based on the number of times $estimateQuery$ is called based on the body atoms from $estimateRule$.
- costOfComputing(***Est4***) is calculated based on the cartesian product on all the answers obtained from all the body atoms. They are summed up for all executed rules.
- numberOfResults(***Est3***) are summed up based on facts obtained from function call to $estimateQuery$ for every rule.
- The numberOfRules(***Est5***) are stored in a data structure, where rules are pushed every time $estimateRule$ is called for a query γ.

Once the estimators are identified, they can be returned to the caller

function . Now that we have the means to obtain values for the estimation features, we need to identify a strategy for deciding between top-down or bottom-up.

Chapter 5

Top Down or Bottom up

As per previous chapter, we have identified strong heuristics for a good prior estimation of the amount of reasoning for answering the input query. In this chapter we determine how we can use these heuristics to decide between top-down approach or bottom-up , i.e., to decide between QSQ or MS.

We have 7 features for each query and we want to give a binary answer (QSQ or MS). This is modelled as a binary classification problem[2]. In statistics, classification is the problem of identifying to which set of categories a new observation belongs to, on the basis of a training set of data containing observations whose category membership is already known. Binary classification is when the there are just two categories involved.

To define our binary prediction model(PM), considering X to be the collection of estimation features on the query and Y indicating if QSQ or MS is favoured, the goal of our binary classification is to build a rule to predict Y given X using only the data at hand.

For every query that is a part of the training data, the decision Y is based on which algorithm performs faster. The same query would be evaluated on both QSQ and MS , and for the same results whichever has lower execution time would be classified as the decision. Once we have a the training data set with the 7 features(X) and the decision (Y), scatter plots are created for every feature against the decision. This helps determine the threshold values for each feature and create the rules that are required for the binary classification model.

Type	Example	Description
Generic Query	RP1(A,B)	Predicate Information is known, but A and B are variables. Hence all records that match the predicate are fetched.
Variable Subject Query	RP1("Alice",B)	Predicate Information is known, A is constant and B is variable. It fetches all records for which Alice is related with the predicate RP1.
Variable Object Query	RP1(A,"Bob")	Predicate Information is known, A is variable and B is constant. It fetches all records for which Bob is related with the predicate RP1.
Boolean Query	RP1("Alice","Bob")	Predicate Information is known, A and B are Constants. It returns TRUE if Alice and Bob are related by predicate and FALSE if not.

Table 5.1: Table classifying different types of atomic queries used.

5.1 Query Generation

As a part of this thesis, the scope of queries are atomic queries and hence all the training queries generated are atomic in nature.

Lets represent a general view of a query as : $RP_0(A, B)$. Here the RP_i maps to a predicate/relation information. A and B can be variables or constant values. This leads us to the different variations of input queries. Table 5.1 discusses the same in length.

To better understand the relationship, lets take an example. Considering working on the LUBM dataset and the usage of the LUBM_LE1 ruleset, there would be multiple RP rules that would map to a predicate information . The below sample rule :

```
RP0(A,B) :- TE(A,<http://www.lehigh.edu/ zhp2/2004/0401/uni
v-bench.owlcolleagues>,B)
```

identifies that RP0(A,B) maps A,B based on predicate $colleagues$ and thus the below query :

```
RP0(<http://www.Department11.University90.edu/AssistantProfessor10>,B)
```

would fetch all the data who are collegues with $AssistantProfessor10$.
The same logic can be extended all the other types of queries.

5.1.1 Manual query creation

The training dataset for evaluation is initially created by manually analyzing the database D and the ruleset R . Restating from before, D would contain a list of RDF triples and R will contain RP rules.

Algorithm 5.1: Creation of training sets manually

```
Data: Dataset D and Ruleset R
Result: Training set of Queries Q
Initialize Input1 = Random subject info from the RDF triples in D.
 Initialize Input2 = Random object from the RDF triples in D.
 for every r in R do
    if predicate of r in D then
        Create Q1 of form RP(A,Input1)
        Create Q2 of form RP(Input2,B)
        Create Q3 of form RP(Input1, Input2)
        if Q1,Q2,Q3 not in Q then
            Add Q1,Q2,Q3 to the queryset Q
        end
    else
        continue
    end
end
```

To manually create a training set for a given D, the first step is to decide on a ruleset to use. Once the ruleset is decided, we would obtain access to the RP rules. The exhaustive list of RP rules stems from the different predicates involved in the RDF triples. The dataset would have the complete data of RDF triples. Since there are 3 types of atomic queries we are working on, for every predicate we create 3 manual queries : boolean queries, variable queries with object information known and variable queries with subject information known. This helps in creating a complete set of queries that can span all scenarios.

Algorithm 5.1 provides a walk-through of the queryset creation. From D, we select subject and object data. For every RP rule in R, create 3 queries with the subject and object data. If these queries are not already present in the Q, add these to Q. The result would be a text file, containing the training dataset.

To span larger dataset of queries and to create them in shorter duration, automation of the task of query creation as mentioned below, was done as a part of a research paper presentation with the help of another team member, Unmesh Joshi, a PhD Scholar at Vrije University,

Amsterdam.

5.1.2 Query Generation

We generate atomic queries from either materialized files or rules file. In both cases, we use database to query and gather some metadata about queries.

Query Generation from materialized rule files

These atomic queries are generated using all predicates and their materialized results at our disposal. For example, consider a binary predicate *memberOf*. Possible results for this predicate are

- $< John, CompSciDept >$
- $< John, VU >$.

Note that the second result (John is member of VU) can be deduced at a later step by the algorithms because it needs to apply more rules. In this case, the knowledge of John being member of Comp Sci Dept and the knowledge of Comp Sci Dept is a member of VU must be combined to give the desired result. Results that are obtained at later steps are marked in the materialized files. Let us assume that the materialized file for the *memberOf* predicate contains 100 records. Out of these 100 records, 10 were derived at step n, 20 were derived at some step $m > n$ and so on. Higher the step count, more is the difficulty of deduction.

We discussed the variation of atomic queries in 5.1. Now the type of the query is determined by two factors :

1. Variation of query (Generic queries start from 100, booleans from 1000)
2. Step count of results (can be 1,2,3,4, and so on)

If a record $< X, Y >$ is obtained at step count 2, then we can craft four queries as follows (assuming it is the result of a binary predicate RP1):

- RP1(A,B) : Generic query with type 102
- RP1(X,B) : Variable query with type 2
- RP1(A,Y) : Variable query with type 2
- RP1(X,Y) : Boolean query with type 1002

Algorithm 5.2: Routine for generating hash table to aid in generation of queries.

```
function estimateRule(ruleFile)
    for ∀rule ∈ ruleFile do
        currentPredicate = Predicate of rule
        if currentPredicate ∉ hashTable then
            hashTable[currentPredicate] = list([atom, variablesMap])// Initialize
        end
        for ∀atom ∈ body − of − rule do
            newPredicate = predicate of atom
            if variables(newPredicate) ≠ variables(currentPredicate) then
                // Do not add this to the list
                continue
            end
            hashTable[currentPredicate].list.add += (atom, variablesMap) if
            newPredicate is extensional predicate then
                hashTable[currentPredicate].indexOfExtPredicate = i
            end
        end
    end
    return hashTable
```

5.1.3 Query Generation from rules file alone

For huge databases, materialization of rules may take a long time. It is essential that training queries generation should be fast, because we need them to do good prediction quickly. Without materialization, we propose an innovative way to generate meaningful queries with database and rules file at our disposal. Based on algorithm 5.3, the procedure to generate training queries from the rules file follows:

The input rule file could contain several rules with the same head. To manage this, we create a hash table using algorithm 5.2, in which the key is the rule heads and the value is an array of all possible atoms that appeared in bodies of the rules with this head. We maintain the index of extensional predicate so that we can quickly access it for each predicate-head without having to go through the array of atoms.

Once we have generated a hash table for the rule file, we begin by going through every key in the hashtable and prepare a query with the *extensional* predicate. If we fire such a query at vlog, it would not in-

duce complex reasoning and we can quickly get an array of results. For a binary predicate, this array would contain pairs of constants from the database. The call to VLog is running with QSQ as default algorithm. If for the input query, results are produced, they can be used to generate queries for the *targetPredicate*. *generateQueries* takes the rulesmap for *targetPredicate* and the results produced to make use of the constants from the results to make subject bound, object bound and boolean queries. This is returned as *resultQueries* If an *extensional* predicate is not present in the database, then we find other atoms for this predicate. Some of these atoms will also appear as head of other rules. This leads a recursive call to the *exploreAllRules* while updating the level of recursion. The atoms of the variable matching map are maintained when the jump to the hash table is being performed. Note that for a rule

$$R1(A, B) : -R2(B, X)$$

$R2$ has only one variable in common with $R1$. If we will find an *extensional* predicate of $R2$ that exists in the database,this would give us a query that would give us some results from the database. Now, this result would contain constants from database that we can use to fabricate the queries for predicate that did not exist in the database.
As an illustration, consider a university database. This database does not have a *colleagueOf* predicate. But a rule says that if an Dan *worksAt* VU and Ben *worksAt* VU, then they are colleagues. Thus, results of a query with *worksAt* predicate would generate some records that will help us fabricate queries for *colleagueOf* predicate.

Algorithm 5.3: Routine for generating queries by deducing rules.

```
Input: hashTable, recurseLevel, vmTarget, targetPredicate, vmPrev, prevPredicate,
Input: vmCur, curPredicate, resultQueries
function exploreAllRules()
    if (level > MAXLEVELS) then
        return
    end
    workingPredicate = hashTable[curPredicate].list[indexOfExtPredicate]
     results = vlog(workingPredicate)
     if len(results) ≠ 0 then
        resultQueries+ = generateQueries(results, variablesMap(targetPredicate)
         return
    end
    foundUsefulPredicates = false
     for ∀atom ∈ hashTable[curPredicate].list do
        atomPredicate = getPredicateFromAtom(atom)
         workingPredicate = hashTable[atomPredicate].list[indexOfExtPredicate]
         results = vlog(workingPredicate) if len(results) ≠ 0 then
            resultQueries+ = generateQueries(results, variablesMap(targetPredicate)
             foundUsefulPredicates = true
             return
        end
    end
    if notfoundUsefulPredicates then
        // Recursive step
        for ∀atom ∈ hashTable[curPredicate].list do
            workingPredicate = getPredicateFromAtom(atom)
             vmWorking = variablesMap(workingPredicate)
             if targetPredicate == prevPredicate then
                updateVariablesMap(vmTarget, vmWorking)// Update according
                   to matching variables of working predicate
            end
            prevPredicate = curPredicate
             curPredicate = workingPredicate
             vmPrev = vmCur
             vmCur = vmWorking
             resultQueries+ = exploreAllRules(hashTable,
             level + 1, vmTarget, targetPredicate, vmPrev,
             prevPredicate, vmCur, curPredicate, resultQueries)
        end
    end
```

Chapter 6

Evaluation

In this chapter we perform evaluation on the algorithms we have proposed and report an analysis of the performances . Section 6.1 reports the setup of the environment in which we conducted the evaluation. Section 6.2 reports the results obtained from the evaluation and their performances.

6.1 Experiment Settings

For the evaluation of the programs we have used the cluster DAS-5 (The Distributed ASCI Supercomputer 5)[1] at the Vrije Universiteit.

DAS-5 includes roughly 200 dual-eight-core compute nodes with 64GB of memory and 128TB of hard disk. The nodes are interconnected through Gigabit Ethernet. We used a machine in DAS5 cluster with Intel(R) Xeon(R) CPU 2.60GHz running CentOS Linux version 7.2.1511. VLog software was compiled with GNU make and we used Python 3.4.5 to run scripts. The datasets we chose are:

- LUBM-1K [8] with 133M triples and used the *LUBM_L* ruleset with 182 rules
- DBPedia dataset[3] with about 1B triples and used the *DBPedia_L* ruleset with 9396 rules.

[1] http://www.cs.vu.nl/das5/

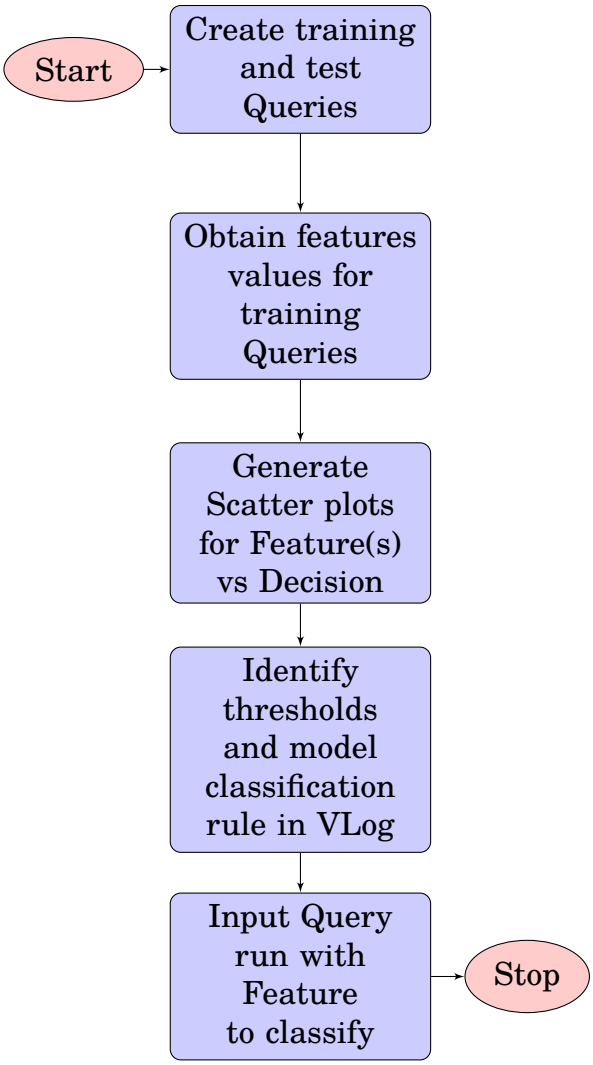


Figure 6.1: Overview of Experiment Process

6.2 Results

We have structured this section in parts based on Figure 6.1 which depicts the algorithms in the order they are run and experiments performed. Subsection 6.2.1 reports the performance for training query creation. Subsection 6.2.2 reports the performances and an evaluation

	Dataset Name	
	LUBM	DBPedia
Number of Training Queries	404 queries	1448 queries
Time taken	9.806 min	8.07 min

Table 6.1: Time taken vs Number of training queries generated for different datasets

of the estimation features algorithm when run against different depth values on VLog. Subsection 6.2.3 depicts the evaluation on the scatter plot creation and threshold identification and subsection 6.2.4 depicts the evaluation of the prediction model.

As per Figure 6.1 the first step in the evaluation is the creation of training queries. This is done based on section 5.1 . Once the queries are generated, values for the estimation features mentioned in section 4 are obtained based on algorithm 4.1 . With the available data , scatter plots are generated to identify relationship between features and decision of whether QSQ or MS takes faster to execute. This relationship is modelled as a rule for a binary classification in VLog.

6.2.1 Training Queries generation

The query generation algorithm is used as a python script that consumes the rulefile to create queries. The usage of hashmap to map the different rules having same head to one key allows us to quickly access each predicate-head without having to go through the array of atoms. What we require from the algorithm is to generate training queries for any input dataset in a reasonable period of time. We have reported in Table 6.1 the training query generation statistics. Row 3 contains the number of training queries generated for LUBM and DBPedia with a standard depth of 4. The 4th row depicts the execution time in minutes. This time involves :

1. The hash table creation for all the predicates in the ruleset.
2. Running Vlog in a top-down mode to generate result records that aids in creating different queries.

	LUBM			DBPedia		
	Depth 2	Depth 4	Depth 8	Depth 2	Depth 4	Depth 8
Number of Training Queries	404	404	404	1448	1448	NA
Time taken	2.687 min	2.698 min	2.836 min	6.700 min	6.907 min	NA

Table 6.2: Performance of estimation feature generation : Time taken across depths

As mentioned earlier, diversity of queries is maintained by creating queries of variable, generic and boolean types via the script.

6.2.2 Feature Generation

The estimation algorithm 4.1 is coded in VLog which fetches value for the estimation features. Since the training queries with the extensional predicate are used, and if the query is run with vlog, it would not induce complex reasoning and facts are derived in lesser time.

A python script is written that executes the estimation algorithm and fetches the features for all training queries into a csv file. These training queries are also executed on both the reasoning approaches, QSQ and MS, and based on which has the lower execution time, the appropriate algorithm for a query is decided. This collection of features and decisions for a set of query helps in identifying a threshold.

We have reported in Table 6.2, the performance of estimation features generation algorithm for different queries on LUBM and DBPedia. The 3rd row represents the depth on which the estimation is done. The 4th row depicts the number of training queries on which the estimation algorithm is run. These queries were generated based on section 5.1. The 5th row depicts the execution time for generating features. This is done on 3 different depths for every query. It is measured from the time the training query file is accessed until all the estimation features are collected and documented into an output file.

From the results it can be identified that for the same number of queries, increasing depth does not lead to a drastic increase in execution times. Infact, for LUBM-1000 and DBPedia the increase in time is in the tune of ms.
We decided to use different depth values while running the experiments as depth signifies the number of iterations that are performed. If it is understood that increased depth could lead to more accurate estimation values, could the accurate estimation improve the efficiency towards prediction?. It can be identified from this section that increased depth does not have a performance impact on feature generation.

6.2.3 Threshold Identification

For identifying threshold values across estimation features, we divide the training queries based on *Est_1* and *Est_2* for each depth across any given dataset:

- Type 1(T1) : Generic Queries.
- Type 2(T2) : Boolean, Subject and Object based queries.

This demarcation is done because, generic queries require far more computation than other queries and grouping both types together to fix the threshold compromises its effectiveness.

In order to identify threshold values for the estimation features, we launch a script that takes as input a csv file. This file contains the features and values for the training queries. It contains 6 fields in the order:

1. numberOfResults
2. costOfComputing
3. numberOfRules
4. numberOfQueries
5. numberOfUniqueRules
6. Decision

This script generates scatter plots between the individual estimation features (1-5) and the Decision (QSQ or MS). We have provided the code snippet that graphs the scatter plot .

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import sys

def plotScatter(estField):
  plt.scatter(estField, Decision, color='m', label= estField + '
      vs Decision')
  plt.xticks(np.arange(min(estField), max(estField)+1, (((max(
     estField)+1) - min(estField))/6)))
  plt.xlabel(estField)
  plt.ylabel('Decision')
  plt.legend()
  name = estField + "Decision.png"
  plt.savefig(name, dpi=300)
```

Listing 6.1: Function that generates scatter plot for individual estimates vs Decision

The script reads the input csv file and maps all the columns into lists. There would be five lists created.
The $plotScatter$ function is called once for every list . First we specify a scatter plot needs to be plotted between the input field and decision field. We specify the range of ticks to lie between min and max value of input fields. We can fix x-axis and y-axis label names and save the generated plot as a png image.

We have represented 2 scatter plots 6.2a and 6.2b generated for LUBM and DBPedia. These 2 are randomly selected for representation and explanation. LUBM is run with 3 depths (2,4 and 8) and DBPedia run with 2 depths (2 and 4), and 5 features are used. Across two sections of queries(T1 and T2), there are 50 scatter plots generated.

$$(((3 * 5) + (2 * 5)) * 2) = 50$$

6.2a is plotted for LUBM where the queries are run with a depth of 4. It is plotted with numberOfResults($Est3$) on the X-axis and decision on the Y-axis. It can be seen that as the $Est3$ increases, MS is favoured over QSQ. To be precise, if $Est3$ is greater than 2242257, MS is selected as the algorithm.
6.2b is plotted on DBPedia. The queries were run with depth 2. X-axis

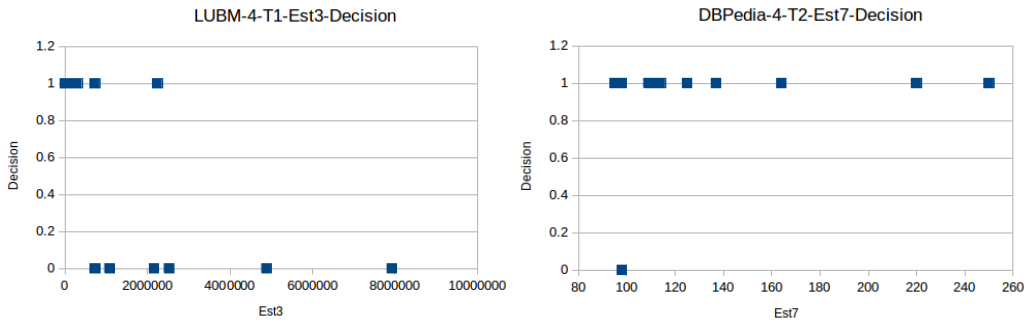


Figure 6.2: Scatter plots for LUBM and DBPedia

	LUBM			DBPedia	
	Depth 2	Depth 4	Depth 8	Depth 2	Depth 4
Est3	*720628*	*2242257*	7924765	*3204*	*5057074*
Est4	*720628*	1081230	1081230	*3435*	*1.00814* * 10^{12}
Est5	4	9	9	61	8337
Est6	*4*	10	10	*9*	*1096*
Est7	*4*	*4*	*4*	*45*	*114*

Table 6.3: Threshold values across datasets for various depths - Type 1 queries

	LUBM			DBPedia	
	Depth 2	Depth 4	Depth 8	Depth 2	Depth 4
Est3	694	1280	2592	56	2504048
Est4	694	1280	2592	493	4.6334 * 10^{11}
Est5	16	36	31	61	8337
Est6	7	27	28	6	1096
Est7	16	16	16	29	98

Table 6.4: Threshold values across datasets for various depths - Type 2 queries

has numberOfUniqueRules($Est7$) plotted while Y-axis has the decision. It can be easily identified that higher $Est7$, QSQ is favoured . If the cost increases greater than 98, QSQ is automatically the favoured algorithm.

Appendix A.2 contains detailed information on all scatter plots generated for LUBM and DBPedia. Based on this information, we have created Table 6.3 and 6.4 with threshold values. Italicized text in cells depict that all values greater than given value, MS is predicted. Normal text in cells depict that all values greater than give value, QSQ is predicted.

These threshold values are coded into VLog as rules. A query can be executed, selecting a single feature based on which algorithm can be predicted. This constitutes our prediction model(PM).

6.2.4 Evaluation of prediction model

We evaluate the prediction model(PM) that we have generated in three aspects :

- Execution time
- Time gain
- Accuracy & Coverage

We create a dataset of 500 test queries that are generated from the full materialization of datasets. For these queries, we already know the faster algorithm. Now, we run these queries against our model to predict which algorithm should be run (QSQ or MS).

Execution time

The PM will be inefficient if the time taken to predict the algorithms takes longer than the actual execution of the algorithm via QSQ or MS. To portray the efficiency obtained by our model, we have run all the queries with QSQ and then with MS and compared the time taken for prediction with the model across all features.

Table 6.5 reports the comparison between QSQ, MS and prediction model(PM). The execution time mentioned in the table is an average time per query, calculated depending on the total number of test queries and the total time taken for execution. What we can see from the results

	Execution time (sec.)					
	LUBM			DBPedia		
	QSQ	MS	**PM**	QSQ	MS	**PM**
Depth-2	41.23	21.32	0.32	30.43	6.67	2.12
Depth-4	48.25	21.32	0.35	33.21	8.92	2.19
Depth-8	47.32	21.82	0.38	NA	NA	NA

Table 6.5: Performance of prediction model against QSQ and MS across Datasets .

is that our prediction model's timing performance beats both QSQ and Magic sets for LUBM and DBPedia dataset. It can also be identified that prediction time increases as depth increases. This behaviour is expected as greater depth involves higher number of iterations. But, this behaviour would be justified if the accuracy of prediction also increases with depth.

Time Gain

Though 6.5 depicts average execution time for the algorithms across all queries, it is identified via analysis that if MS is faster, it is faster by 11 times(avg. time diff) and when QSQ is faster it is faster by 6 times(avg.). Hence its of importance that PM predicts the correct algorithm to obtain higher efficiency.

The usefulness of the prediction model we have built depends upon whether it can provide a substantial time gain against the existing approach.

Tab. 6.6 reports the execution time which would take if we only execute QSQ, only MS, or if instead we decide the algorithm at runtime. Notice that these experiments were performed only on the queries that were eligible for prediction (i.e., estimator value greater than first threshold). Each line reports the total runtime on all queries for every estimator. From the table, we observe that only *EF1* and *EF4* lead to an increase of the performance. For all other estimators, the strategy

Estimator	QSQ	MS	**Prediction Model**
Est1	18.6	11.97	9.3
Est2	40.02	4.36	30.6
Est3	41.32	5.06	33.76
Est4	9.8	2.4	2.1
Est5	41.32	5.06	33.76

Table 6.6: Time gain for PM(min.)

of always selecting MS would lead to better runtimes. However, notice that using any estimator is better than using only QSQ.

Accuracy & Coverage

To evaluate the efficiency of prediction, we use coverage and accuracy. With respect to this thesis, we define accuracy as :

$$Acc_{PM} = \frac{N_{Correct}}{T_{Satisfied}}$$

where
Acc_{PM} = Accuracy of the Model
$N_{Correct}$ = Number of queries predicted correctly.
$T_{Satisfied}$ = Total number of queries that satisfy threshold condition.
and Coverage as :

$$Cov_{PM} = \frac{T_{Satisfied}}{T_{Queries}}$$

where
Acc_{PM} = Accuracy of the Model
$T_{Satisfied}$ = Total number of queries that satisfy threshold condition.
$T_{Queries}$ = Total number of test queries .

Taking an example to explain:
Let us consider 100 test queries created for LUBM_1000. For each of these queries we know the better algorithm for execution(QSQ or MS). We perform the prediction based on *Est1*. We identify that 70 queries

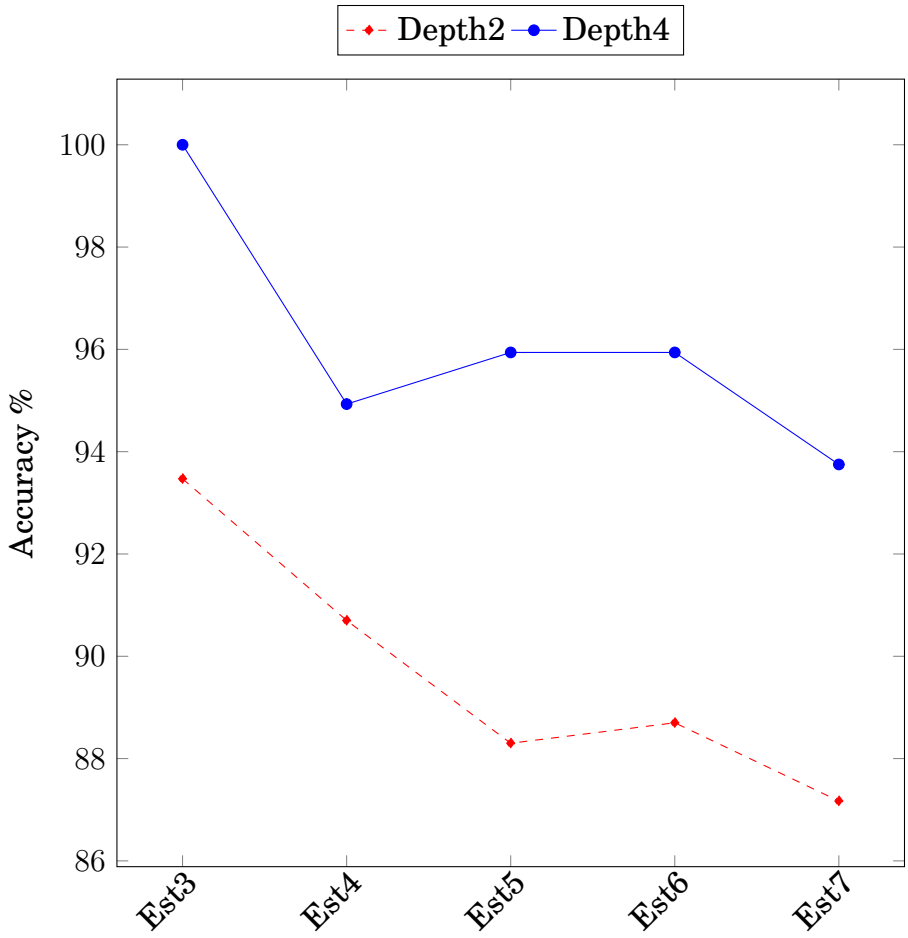


Figure 6.3: Accuracy of DBPedia dataset

have value greater than threshold that's fixed for *Est1*,so the coverage is 70/100. Out of these 70 queries, PM predicts the correct algorithm for 50 of them,so accuracy is 50/70.

We report from Fig. 6.3 and 6.4 , the accuracy of the prediction model across the 5 different features for LUBM-1000 and DBPedia dataset. The experiment performed answers the question : *Does increasing the depth result in increased accuracy of prediction?*. Considering Fig 6.3, the X-axis depicts the 5 estimation features that are individually used for prediction and whose accuracy is calculated. This graph is plotted for DBPedia for depth 2 and depth 4. It can be noted that for all the 5

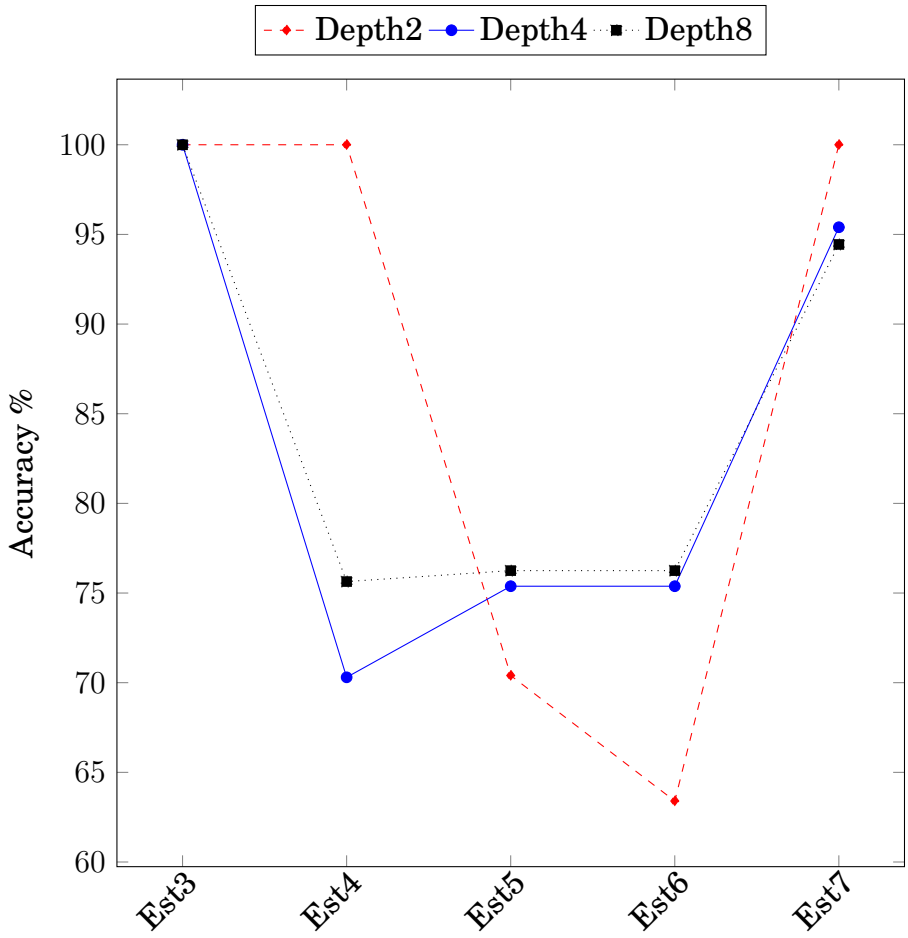


Figure 6.4: Accuracy of LUBM(1000) dataset

features as depth increases, the accuracy of prediction increases. Considering Fig 6.4, this graph is for the LUBM-1000 accuracy performance. It can be identified that for *Est3*, we obtain 100% accuracy across all depths and for *Est5* and *Est6* as the depth increases, there is a steady increase in the accuracy of algorithm prediction.

The case with *Est4* and *Est7* needs to be elaborated. Though, there is a visible accuracy increase from depth 4 to depth 8 for both the estimates, the accuracy of prediction is highest for depth 2. On further analysis it can be identified that for lower depths the prediction based on single estimation feature can be tainted by the influence of other fea-

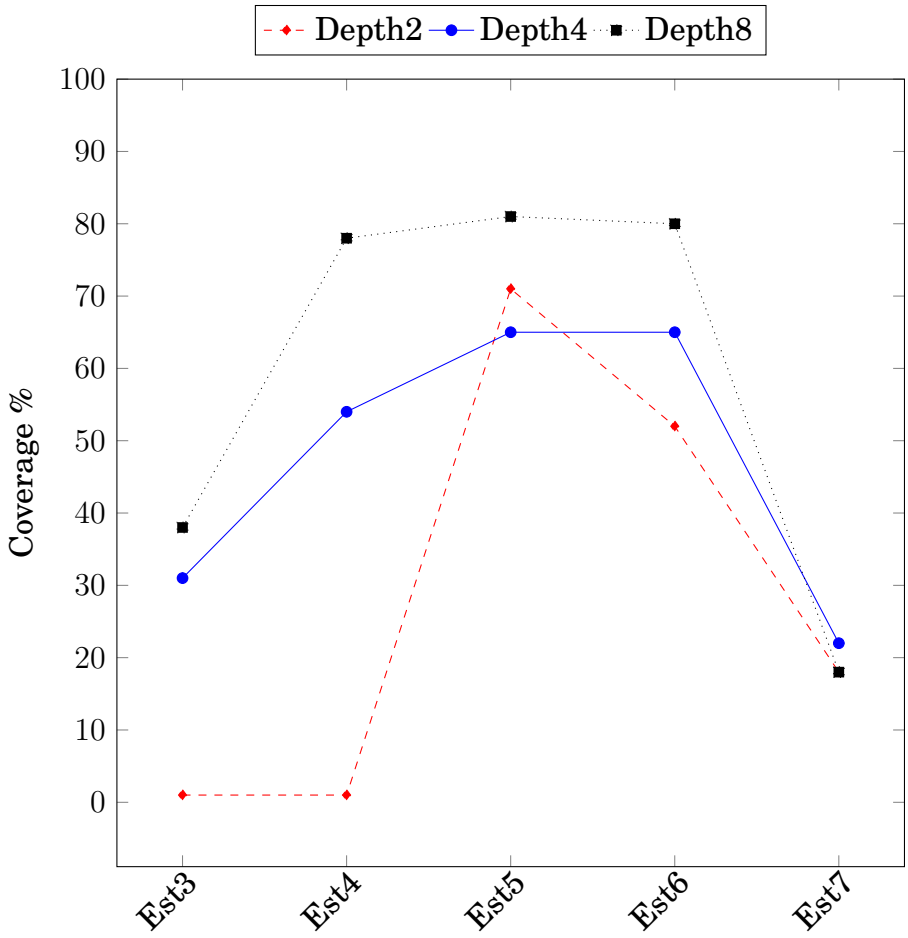


Figure 6.5: Coverage of LUBM(1000) dataset

tures, i.e., 2 different queries having the same value for *Est4*, predicts two different algorithms, based on an influence by an feature that has higher accuracy. This knowledge brings the concept of coverage to the performance analysis. For this thesis, coverage can be defined as the total number of queries that are correctly predicted from the total number of queries that fit the threshold criteria.

We report from Figure 6.5 and 6.6 the coverage for LUBM and DBPedia.It can be noted that for majority of features, as depth increases, the coverage has increased.

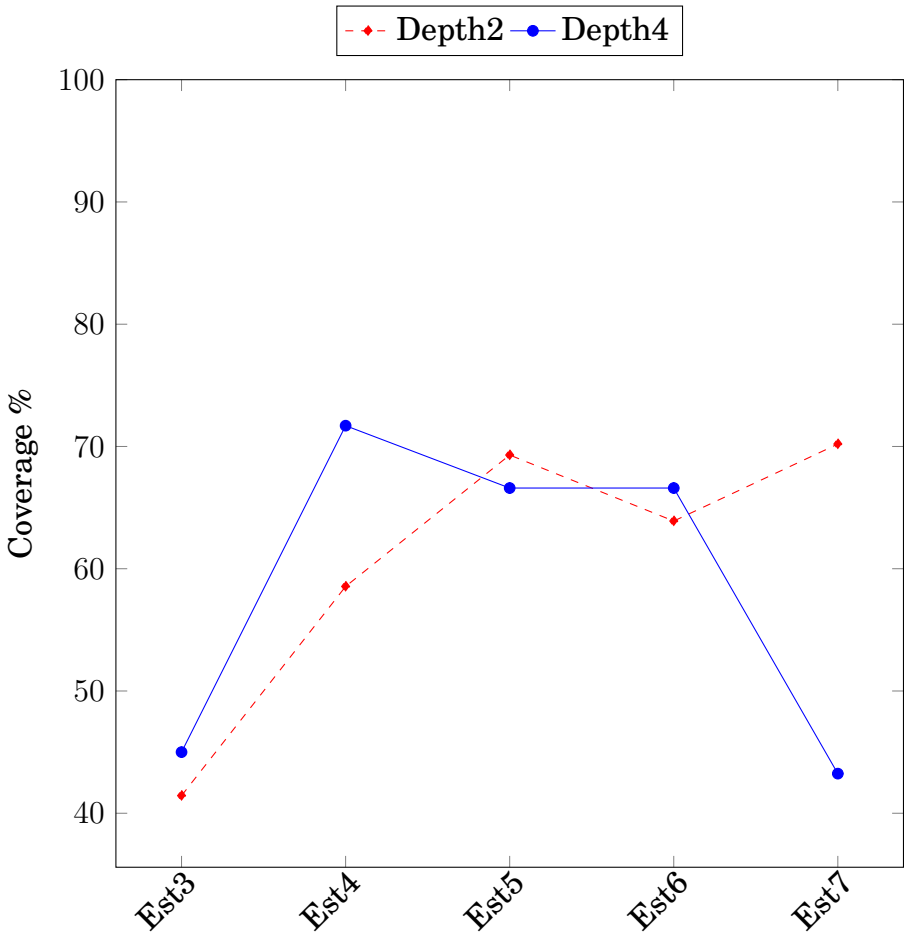


Figure 6.6: Coverage of DBPedia dataset

Looking at the statistics,

For LUBM_1000:

$$Coverage_{minm} == 0\%, Coverage_{maxm} = 81\%$$

and DBPedia :

$$Coverage_{minm} == 41\%, Coverage_{maxm} = 72\%$$

This tells us that we have been able to build a prediction model that can, at best cover 80% of all query scenarios.
Why haven't we been able to cover the remaining 20% ?
On investigation, we identified that single feature estimation can do only so much . There are relationships between multiple features that also affect the decision.

Looking at all the results derived, a quick summary would be:

- There are features that affect the decision of whether to go Top-down or Bottom-up.
- The estimation procedure written is considerably faster than actual evaluation of algorithms.
- Though *Est1* has higher accuracy, the coverage is small. Hence it is better to select a feature that has comparable performance across accuracy and coverage like *Est6*.
- There is an inherant limitation with prediction based on single features. We need to analyze relationships between different features that affect decisions.

Chapter 7

Related Work

In this chapter we will briefly explain some relevant work done in the area so that the reader can place this thesis in context.

Bottom-Up over Top-down : The question of QSQ or MS has been discussed and researched over time . The most significant discussions are [5], which is often misunderstood that for a safe Datalog program the conclusion is that Bottom-Up beats Top-down for all scenarios, but its the case only for a specific case mentioned as Queue-Based Rule/Goal Tree expansion(QRGT) which doesn't consider the arbitrary SIPS (sideways information passing strategies) and [14] where research is done on Prolog.

Materialization : The process of reasoning has strong roots via materialization. Large amounts of RDF data(tune of 15 million triples) were processed via distributed reasoning in MarVIN [13]. WebPIE [21] introduced a scalable approach based on MapReduce implementation of standard inference rules . It caters to dataset with upto 100 billion edges for processing.

Online Reasoning : [7] discusses integrating inference at run time on OpenLinkś Virtouso. Another RDF Engine that performs on demand inferencing is 4s-reasoner [15] peforming backward chained clustered RDFS reasoning in 4store. QueryPIE [20] performs backward chain reasoning for atomic queries on datasets spanning 1 Billion triples.

Chapter 8

Future works and conclusions

In this thesis, we deal with the problem of efficiently performing online reasoning on a given Knowledge graph. Existing research focuses on materialization which is resource and time intensive.

Summary : The research question we tried to answer in this thesis was "Considering the process of Online Reasoning on a KG, is it possible to determine an estimation based strategy, which when given an Input query, will be able to determine which algorithm performs better? QSQR or MS". We tried to answer this question by first performing a thorough analysis of two Query-based reasoning algorithm : QSQ and MS based on Datalog. This helped us in understanding the templates used for storing information during intermediate stages of an evaluation . Based on this analysis, we identified 7 estimation features that impact the runtime (i.e., cost) of reasoning. We proposed an algorithm to fetch values for these features based on the simulation of execution of QSQ up to a certain depth. This simulation mimics the execution of QSQ without actually performing any derivation. In order to perform the prediction based on these features, we propose a binary classification model. We identify an algorithm to generate queries from rules file that recursively crawls through rules and produces meaningful queries. These are training queries via which we modelled rules in VLog for individual features. This classification model efficiently and accurately predicts the algorithm for majority of queries and computes results faster for any given query.

Future Work : Based on the evaluation of our model, we have identified areas of improvement.

The threshold used in defining the classification rule is as strong as the training queries that generate the feature values. We need to improve the query generation algorithm to generate queries that span all feasible query scenarios .
The coverage of certain features is very low. This is because, by them self, these features do not have a pattern in algorithm decision. On further analysis, we have identified that in many query scenarios, multiple features together form a pattern for decision making. So we need to investigate relationships between different combination of features and the decision in order to move to a multi-feature prediction model.

Conclusion : In this thesis, we have answered our research question and shown that it is possible to predict the better reasoning algorithm for a given input query. We have presented 7 estimation features which are helpful to understand the cost of reasoning and hence can be used to improve the runtime of online reasoning. Based on the results it is identified that the PM is able to predict the correct algorithm in considerably smaller time period and has potential though the performance varies across different features.

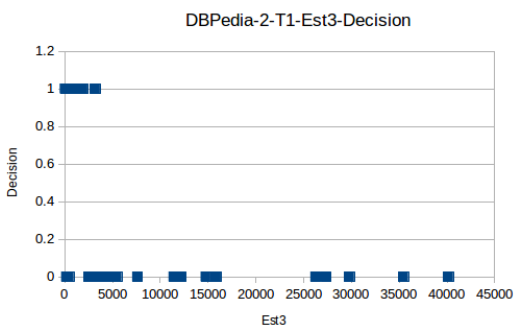
Appendix A

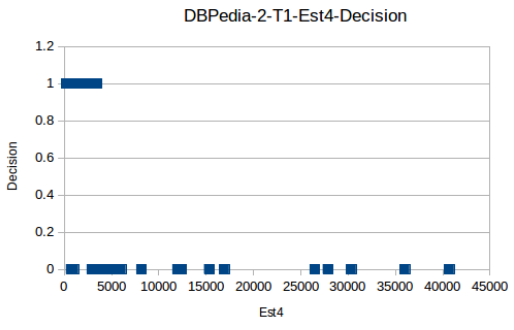
A.1 Naming Convention

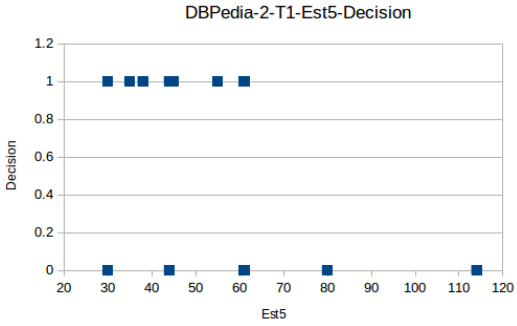
This appendix describes a suggested set of conventions for naming training file, test file and scatter plots used in this thesis :

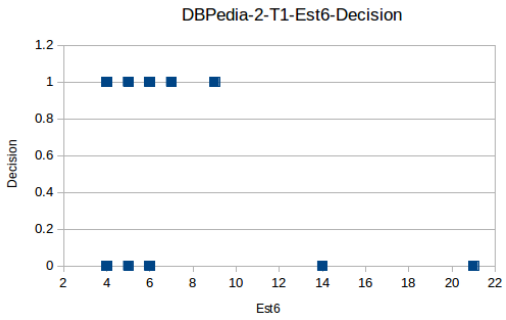
```
(LUBM | DBPedia)_(train | test)_(2 | 4 | 8)
Dataset_TypeOfFile_Depth

(LUBM | DBPedia)_(2 | 4 | 8)_(T1 | T2)_(Est3 | Est4 | Est5
| Est6 | Est7)_Decision
Dataset_Depth_TypeofQuery_EstimationFeature_Decision
```

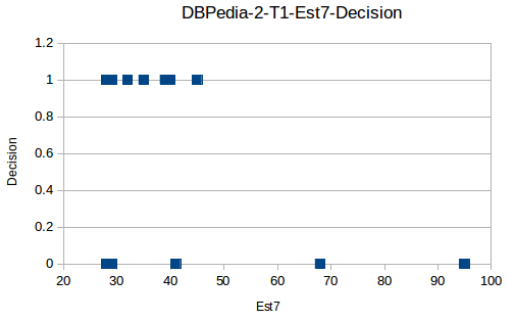
A.2 Scatter plots

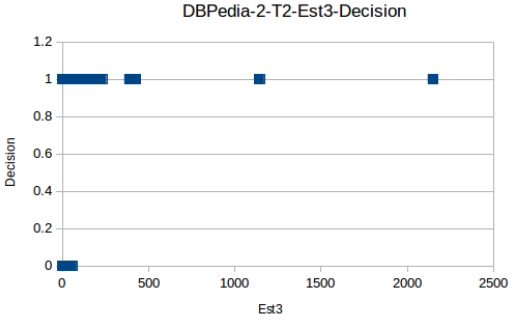


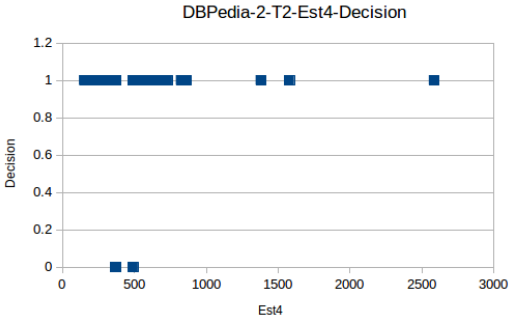


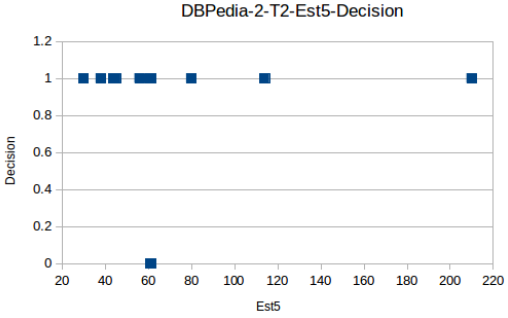

DBPedia-2-T1-Est5-Decision
Decision
Est5

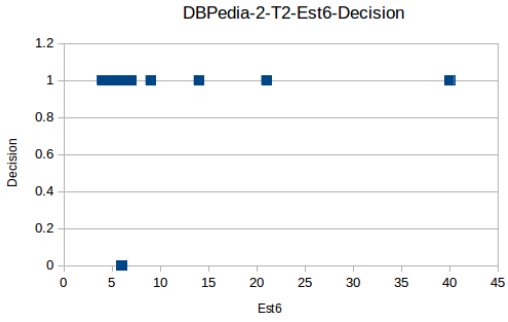

DBPedia-2-T1-Est6-Decision
Decision
Est6

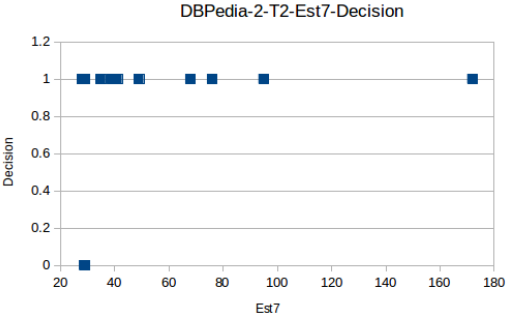

DBPedia-2-T1-Est7-Decision
Decision
Est7

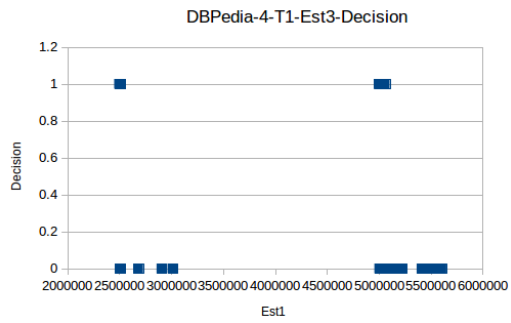

DBPedia-2-T2-Est3-Decision
Decision
Est3

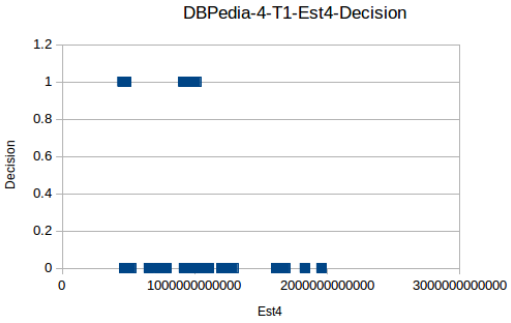

DBPedia-2-T2-Est4-Decision
Decision
Est4

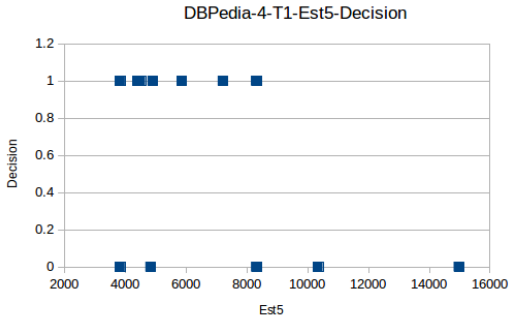

DBPedia-2-T2-Est5-Decision
Decision
Est5

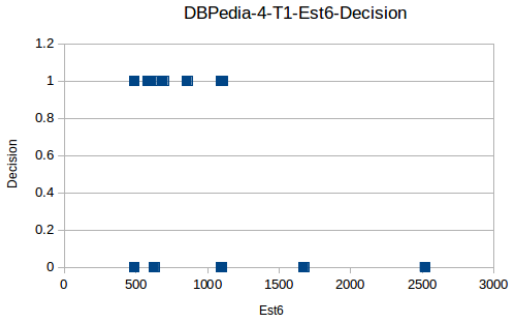

DBPedia-2-T2-Est6-Decision
Decision
Est6

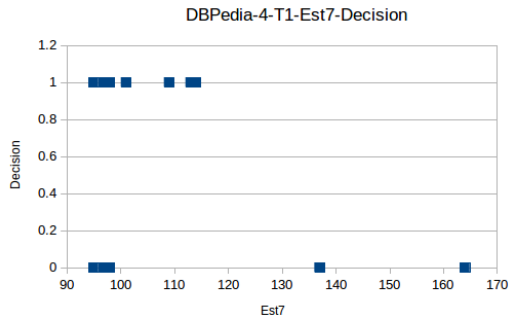

DBPedia-2-T2-Est7-Decision
Decision
Est7

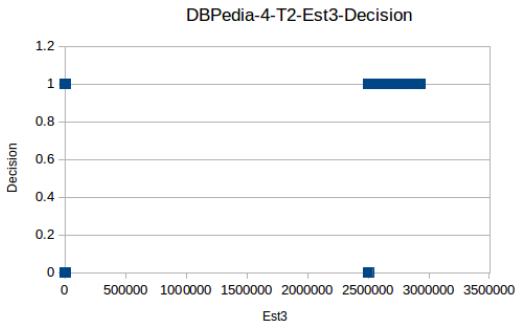

DBPedia-4-T1-Est3-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
2000000 2500000 3000000 3500000 4000000 4500000 5000000 5500000 6000000
Est1

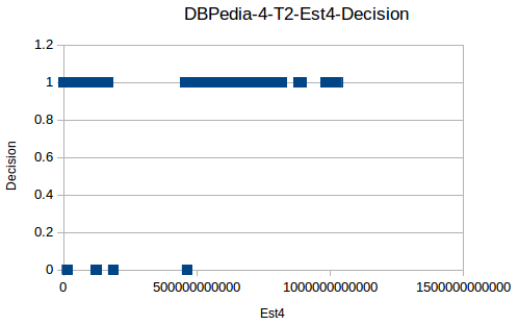

DBPedia-4-T1-Est4-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
0 1000000000000 2000000000000 3000000000000
Est4

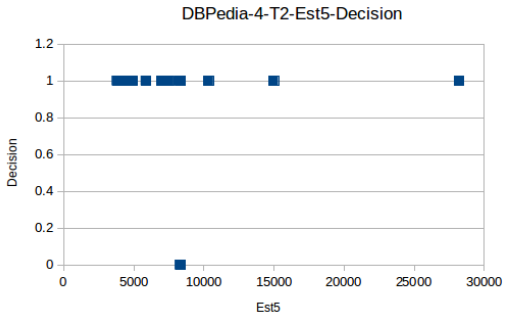

DBPedia-4-T1-Est5-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
2000 4000 6000 8000 10000 12000 14000 16000
Est5

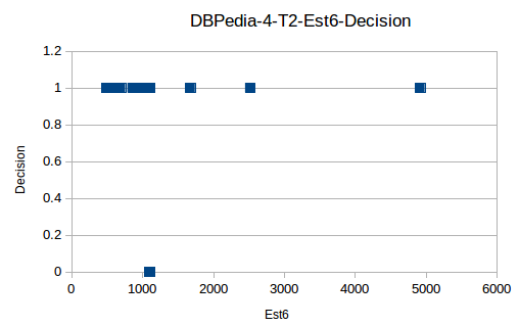

DBPedia-4-T1-Est6-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
0 500 1000 1500 2000 2500 3000
Est6

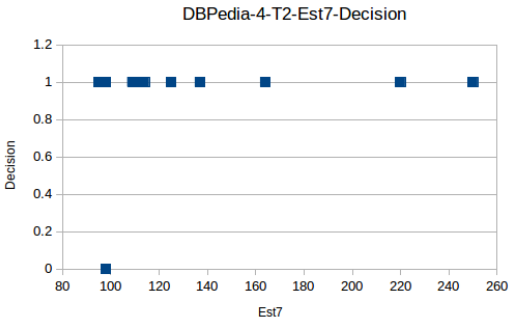

DBPedia-4-T1-Est7-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
90 100 110 120 130 140 150 160 170
Est7

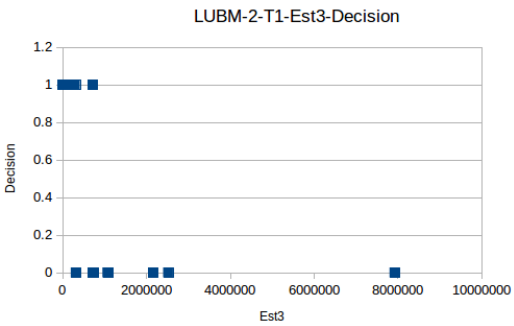

DBPedia-4-T2-Est3-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
0 500000 1000000 1500000 2000000 2500000 3000000 3500000
Est3

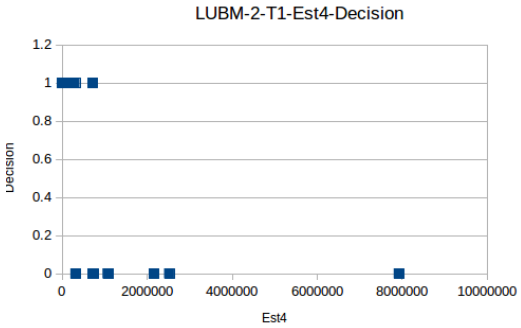

DBPedia-4-T2-Est4-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
0 500000000000 1000000000000 1500000000000
Est4

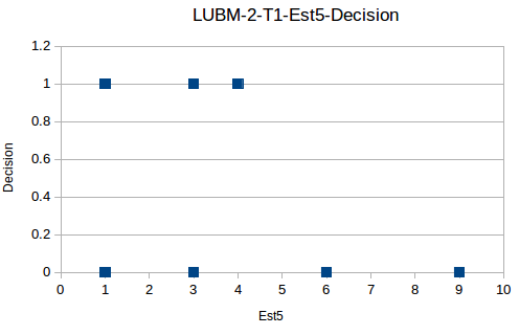

DBPedia-4-T2-Est5-Decision
Decision
1.2
1
0.8
0.6
0.4
0.2
0
0 5000 10000 15000 20000 25000 30000
Est5

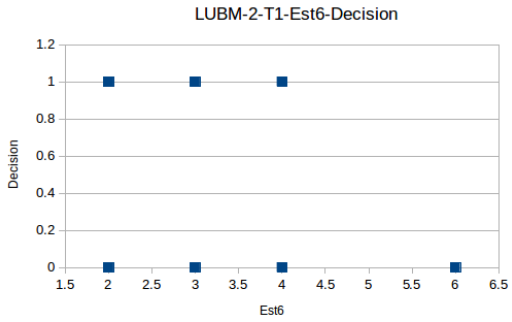

DBPedia-4-T2-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
0
1000
2000
3000
4000
5000
6000

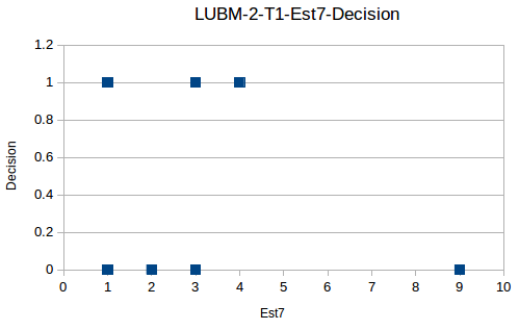

DBPedia-4-T2-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
80
100
120
140
160
180
200
220
240
260

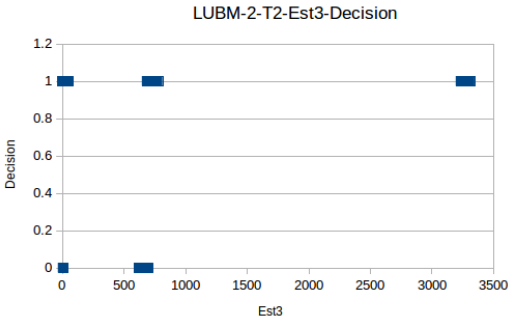

LUBM-2-T1-Est3-Decision
Decision
Est3
0
0.2
0.4
0.6
0.8
1
1.2
0
2000000
4000000
6000000
8000000
10000000

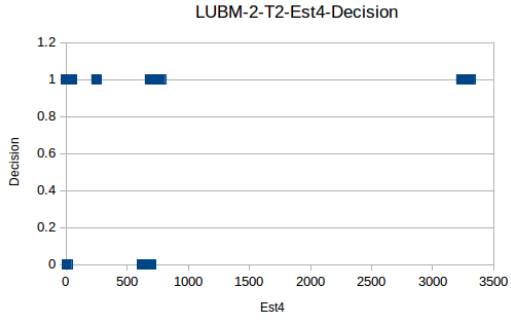

LUBM-2-T1-Est4-Decision
Decision
Est4
0
0.2
0.4
0.6
0.8
1
1.2
0
2000000
4000000
6000000
8000000
10000000

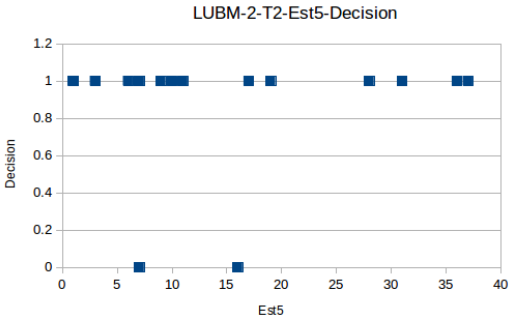

LUBM-2-T1-Est5-Decision
Decision
Est5
0
0.2
0.4
0.6
0.8
1
1.2
0
1
2
3
4
5
6
7
8
9
10

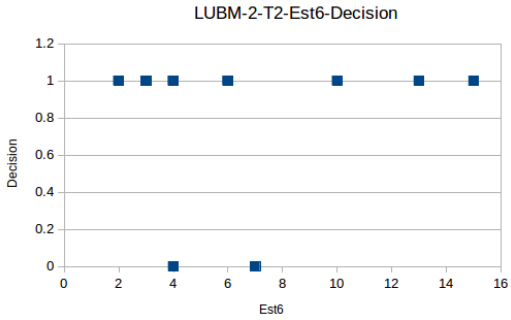

LUBM-2-T1-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
6.5

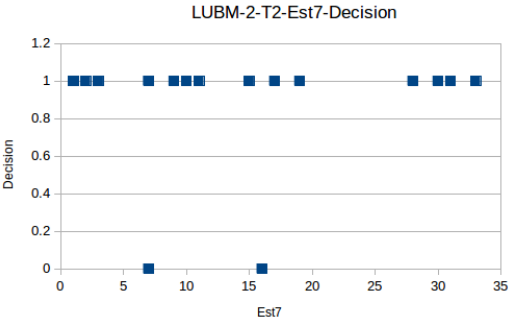

LUBM-2-T1-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
0
1
2
3
4
5
6
7
8
9
10

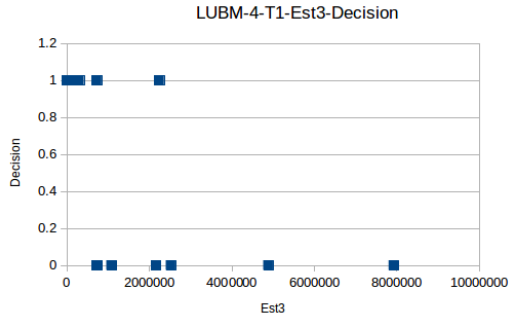

LUBM-2-T2-Est3-Decision
Decision
Est3
0
0.2
0.4
0.6
0.8
1
1.2
0
500
1000
1500
2000
2500
3000
3500

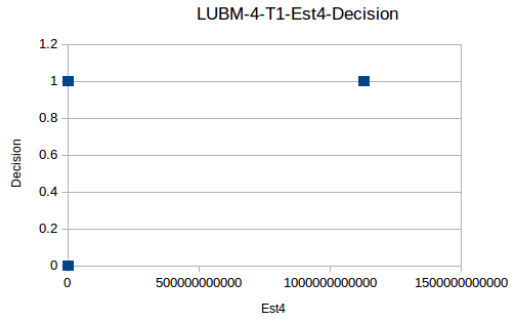

LUBM-2-T2-Est4-Decision
Decision
Est4
0
0.2
0.4
0.6
0.8
1
1.2
0
500
1000
1500
2000
2500
3000
3500

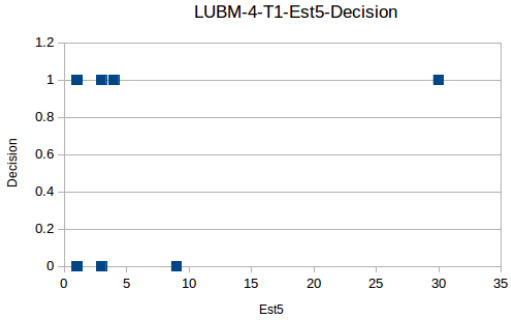

LUBM-2-T2-Est5-Decision
Decision
Est5
0
0.2
0.4
0.6
0.8
1
1.2
0
5
10
15
20
25
30
35
40

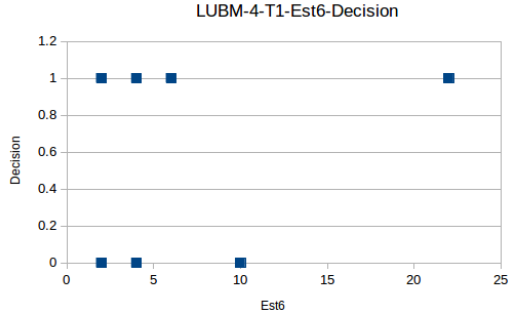

LUBM-2-T2-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
0
2
4
6
8
10
12
14
16

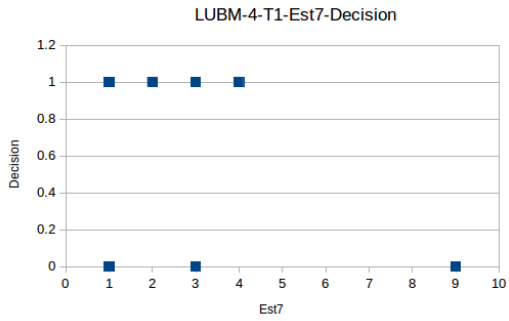

LUBM-2-T2-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
0
5
10
15
20
25
30
35

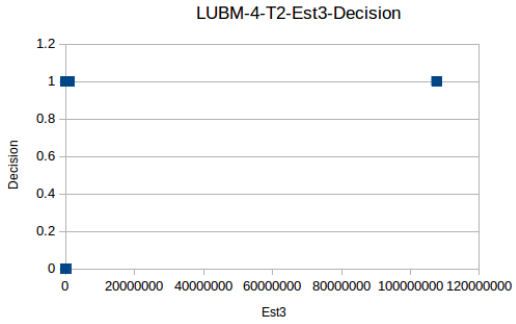

LUBM-4-T1-Est3-Decision
Decision
Est3
0
0.2
0.4
0.6
0.8
1
1.2
0
2000000
4000000
6000000
8000000
10000000

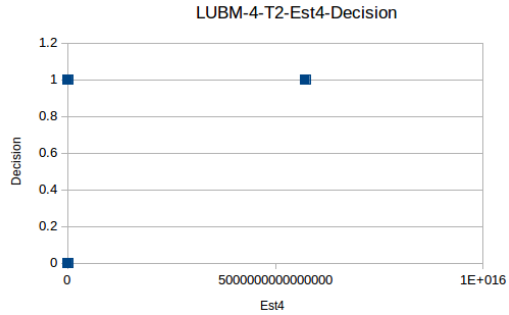

LUBM-4-T1-Est4-Decision
Decision
Est4
0
0.2
0.4
0.6
0.8
1
1.2
0
500000000000
1000000000000
1500000000000

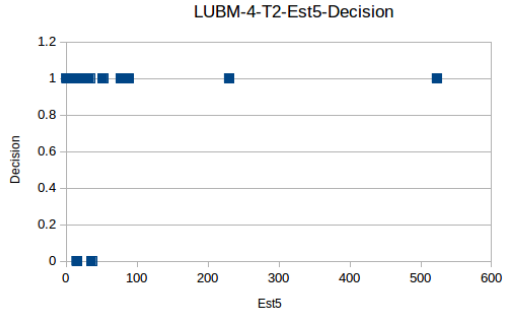

LUBM-4-T1-Est5-Decision
Decision
Est5
0
0.2
0.4
0.6
0.8
1
1.2
0
5
10
15
20
25
30
35

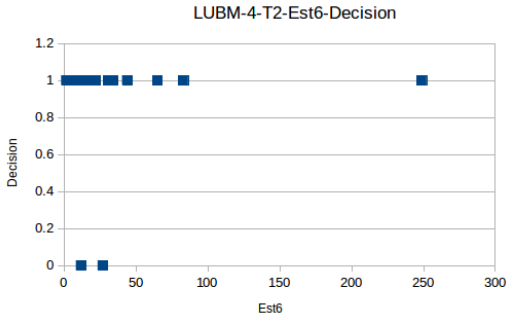

LUBM-4-T1-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
0
5
10
15
20
25

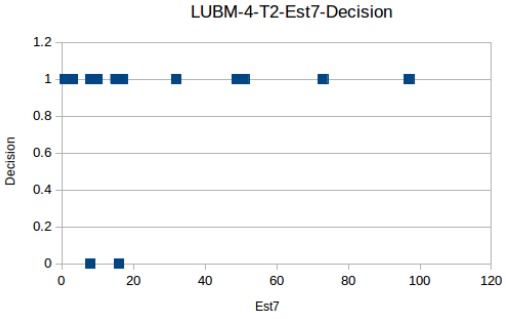

LUBM-4-T1-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
0 1 2 3 4 5 6 7 8 9 10

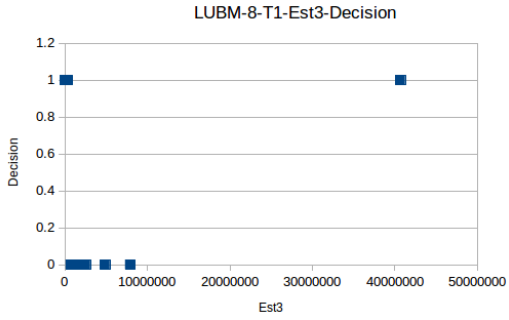

LUBM-4-T2-Est3-Decision
Decision
Est3
0
0.2
0.4
0.6
0.8
1
1.2
0 20000000 40000000 60000000 80000000 100000000 120000000

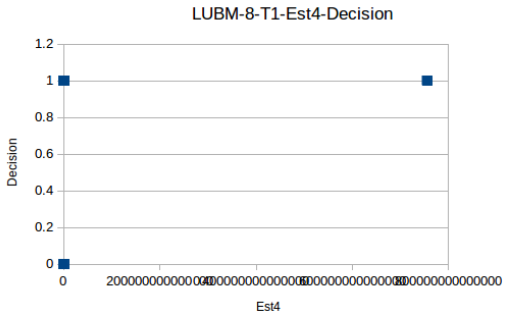

LUBM-4-T2-Est4-Decision
Decision
Est4
0
0.2
0.4
0.6
0.8
1
1.2
0 5000000000000000 1E+016

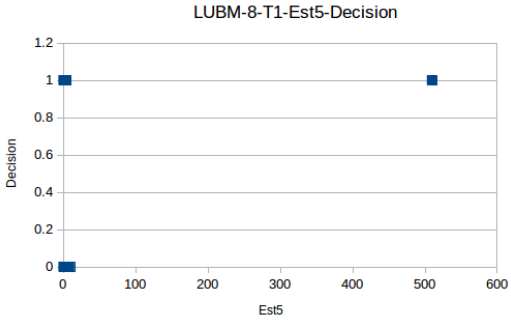

LUBM-4-T2-Est5-Decision
Decision
Est5
0
0.2
0.4
0.6
0.8
1
1.2
0 100 200 300 400 500 600

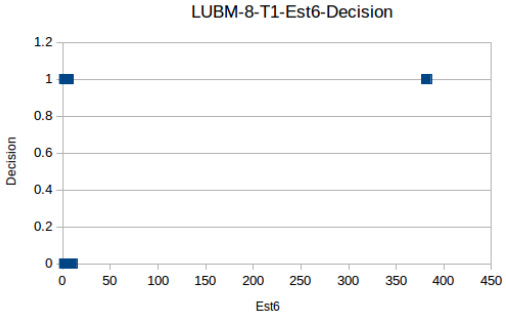

LUBM-4-T2-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
0 50 100 150 200 250 300

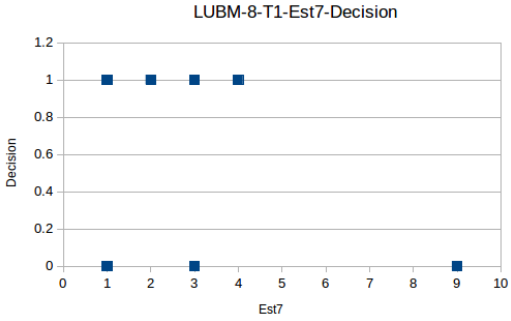

LUBM-4-T2-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
0 20 40 60 80 100 120

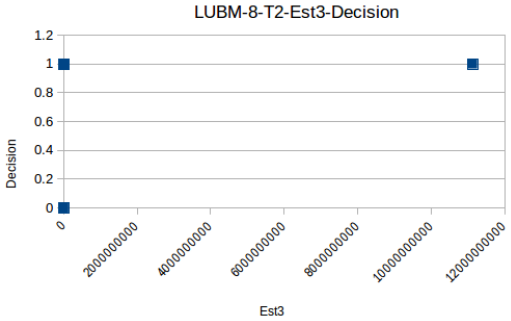

LUBM-8-T1-Est3-Decision
Decision
Est3
0
0.2
0.4
0.6
0.8
1
1.2
0 10000000 20000000 30000000 40000000 50000000

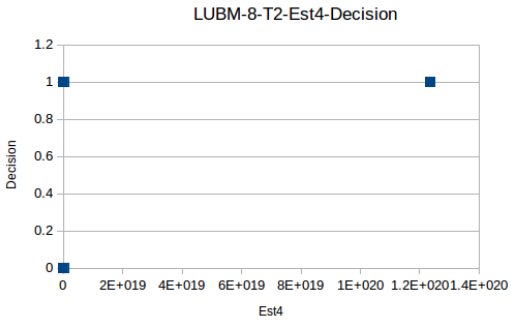

LUBM-8-T1-Est4-Decision
Decision
Est4
0
0.2
0.4
0.6
0.8
1
1.2
0

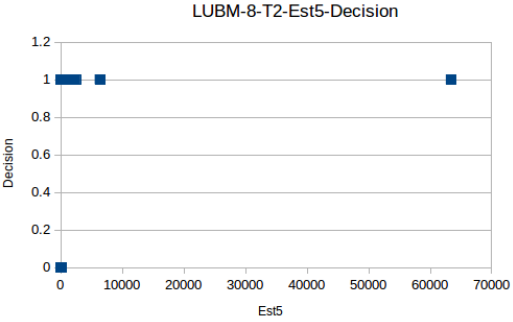

LUBM-8-T1-Est5-Decision
Decision
Est5
0
0.2
0.4
0.6
0.8
1
1.2
0
100
200
300
400
500
600

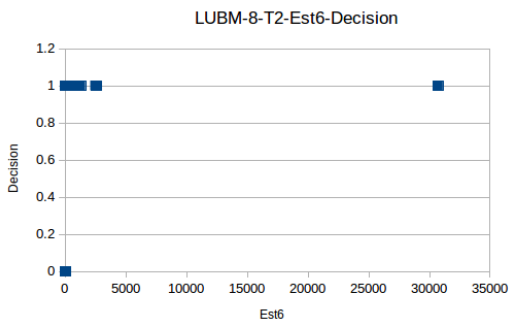

LUBM-8-T1-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
0
50
100
150
200
250
300
350
400
450

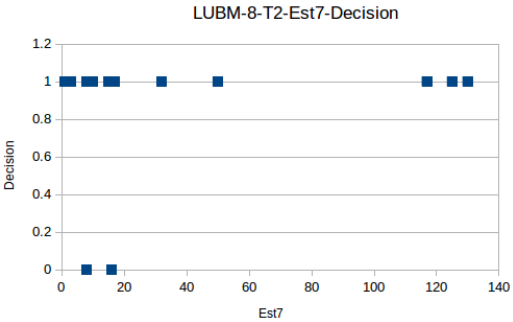

LUBM-8-T1-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
0
1
2
3
4
5
6
7
8
9
10


LUBM-8-T2-Est3-Decision
Decision
Est3
0
0.2
0.4
0.6
0.8
1
1.2
0
2000000000
4000000000
6000000000
8000000000
10000000000
12000000000


LUBM-8-T2-Est4-Decision
Decision
Est4
0
0.2
0.4
0.6
0.8
1
1.2
0
2E+019
4E+019
6E+019
8E+019
1E+020
1.2E+020
1.4E+020


LUBM-8-T2-Est5-Decision
Decision
Est5
0
0.2
0.4
0.6
0.8
1
1.2
0
10000
20000
30000
40000
50000
60000
70000


LUBM-8-T2-Est6-Decision
Decision
Est6
0
0.2
0.4
0.6
0.8
1
1.2
0
5000
10000
15000
20000
25000
30000
35000


LUBM-8-T2-Est7-Decision
Decision
Est7
0
0.2
0.4
0.6
0.8
1
1.2
0
20
40
60
80
100
120
140

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 7, 9, 16).

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014 (cit. on p. 29).

[3] Sören Auer et al. "Dbpedia: A nucleus for a web of open data". In: *The semantic web* (2007), pp. 722–735 (cit. on pp. 9, 37).

[4] Tim Berners-Lee, James Hendler, Ora Lassila, et al. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 28–37 (cit. on p. 5).

[5] "Bottom-up Beats Top-down for Datalog". In: PODS '89. New York, NY, USA. URL: http://doi.acm.org/10.1145/73721.73736 (cit. on p. 53).

[6] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. "Sesame: A generic architecture for storing and querying rdf and rdf schema". In: *International semantic web conference*. Springer. 2002, pp. 54–68 (cit. on p. 6).

[7] Orri Erling and Ivan Mikhailov. "SPARQL and scalable inference on demand". In: *Proceedings of the 6th European Semantic Conference (ESWC2009), Heraklion, Greece*. 2009 (cit. on p. 53).

[8] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2 (2005), pp. 158–182 (cit. on pp. 9, 37).

[9] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. “Datalog and emerging applications: an interactive tutorial”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, pp. 1213–1216 (cit. on p. 14).

[10] Vladimir Kolovski, Zhe Wu, and George Eadon. “Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system”. In: *The Semantic Web–ISWC 2010* (2010), pp. 436–452 (cit. on p. 7).

[11] Frank Manola, Eric Miller, Brian McBride, et al. “RDF primer”. In: *W3C recommendation* 10.1-107 (2004), p. 6 (cit. on p. 5).

[12] Boris Motik et al. “Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems.” In: *AAAI*. 2014, pp. 129–137 (cit. on p. 7).

[13] Eyal Oren et al. “Marvin: Distributed reasoning over large-scale Semantic Web data”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 7.4 (2009), pp. 305–316 (cit. on p. 53).

[14] Raghu Ramakrishnan and S Sudarshan. “Top-down vs. bottom-up revisited”. In: (cit. on p. 53).

[15] Manuel Salvadores et al. “4s-reasoner: Rdfs backward chained reasoning support in 4store”. In: *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*. Vol. 3. IEEE. 2010, pp. 261–264 (cit. on p. 53).

[16] Dmitry Tsarkov and Ian Horrocks. “FaCT++ description logic reasoner: System description”. In: *Automated reasoning* (2006), pp. 292–297 (cit. on p. 6).

[17] Jacopo Urbani, Ceriel JH Jacobs, and Markus Krötzsch. “Column-Oriented Datalog Materialization for Large Knowledge Graphs.” In: *AAAI*. 2016, pp. 258–264 (cit. on p. 7).

[18] Jacopo Urbani, Ceriel JH Jacobs, and Markus Krötzsch. “VLog: A Column-Oriented Datalog System for Large Knowledge Graphs.” In: *International Semantic Web Conference (Posters & Demos)*. 2016 (cit. on p. 17).

[19] Jacopo Urbani et al. “Dynamite: Parallel materialization of dynamic rdf data”. In: *International Semantic Web Conference*. Springer. 2013, pp. 657–672 (cit. on p. 7).

[20] Jacopo Urbani et al. “QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases”. In: *The Semantic Web–ISWC 2011* (2011), pp. 730–745 (cit. on p. 53).

[21] Jacopo Urbani et al. “WebPIE: A web-scale parallel inference engine using MapReduce”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 10 (2012), pp. 59–75 (cit. on pp. 7, 53).

[22] W3C Wiki. *TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics*. [Online; accessed 30-July-2017]. 2015. URL: https://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics (cit. on p. 6).