GRAPH SUMMARIES FOR OPTIMIZING GRAPH PATTERN QUERIES ON RDF DATABASES

by

ANGELA I. MADUKO

(Under the Direction of Amit P. Sheth and John A. Miller)

ABSTRACT

The adoption of the Resource Description Framework (RDF) as a metadata representation standard is spurring the development of high-level mechanisms for storing and querying RDF data. Many of the proposed systems are built on Relational/Object-Relational Databases with a translation of queries posed in the supported RDF query language to SQL for processing by the database. Graph pattern matching which matches a query graph against a data graph, often require join operations. To process join operations, the database optimizer determines an optimal join order from a cost model which employs the expected cardinality of join results as a key parameter. This parameter is estimated from a statistical summary of the data maintained in memory. In this work, we argue that the data summarization technique employed by database systems are oblivious of the graph structure of RDF data and may lead to estimation errors which result in the choice of a sub-optimal query plan. We present and evaluate two techniques for estimating the frequency of subgraphs utilizing a small statistical summary of the graph, based on occurrences. In the first technique, we summarize the graph in the P-Tree by pruning small subgraphs based on a valuation scheme that blends information about their importance and estimation power. In the second technique, we assume that edge occurrences on edge sequences of length maxL are position independent. We then summarize the most informative dependencies in the MD-Tree. In both techniques, we assume conditional independence to estimate the frequencies of larger subgraphs. We present extensive experiments on real world and synthetic datasets which confirm the feasibility of our approach. Our experiments are geared towards showing that the estimates obtained from the proposed summaries are accurate as well as effective for optimizing graph pattern queries posed over RDF graphs.

INDEX WORDS: Frequency estimation, Graph summaries, Data summaries, Statistical Summaries, RDF Join Queries, Graph Pattern Queries

GRAPH SUMMARIES FOR OPTIMIZING GRAPH PATTERN QUERIES ON RDF DATABASES

by

ANGELA I. MADUKO

B.S, The University of Nigeria, Nsukka, Nigeria, 1998

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2009

GRAPH SUMMARIES FOR OPTIMIZING GRAPH PATTERN QUERIES ON RDF DATABASES

by

ANGELA I. MADUKO

Major Professor:	Amit P. Sheth
	John A. Miller
Committee:	Jonathan Arnold
	Budak Arpinar
	Krzysztof Kochut

DEDICATION

I would like to dedicate my doctoral thesis to my parents for their love and support.

ACKNOWLEDGEMENTS

I give thanks to the almighty God for making it possible for me to achieve this highest level of education, by strengthening me and giving me the endurance that is necessary for the pursuit of this level of education. Without Him, none of this would have been possible.

I am greatly indebted to my major advisor, Dr. Amit P. Sheth, for his guidance and support throughout my studentship. His ready availability made it easy to get feedback and direction at different points. I am also indebted to Dr. John A. Miller, my co-major advisor, for his guidance especially in the last stages of my work. His ready availability made it easy to get feedback and direction especially during the last stages of my studentship. I would like to thank my advisory committee members, Dr. Budak Arpinar, Dr. Krzysztof Kochut and Dr. Jonathan Arnold for accepting the task of participating on my doctoral advisory committee. I am especially grateful to Dr. Paul Schliekelman for his ready availability and technical feedback and guidance at different points. Special thanks to my friend Kemafor Anyanwu for her words of advice at different points through out this journey.

I am deeply indebted to my family for the sacrifices that they made to support my pursuit of this degree and the patience they showed throughout the process. This could not have been possible without their love and support. I am also grateful to Chidi Okonkwo for his support during the pursuit of my doctoral degree.

This work was supported in part by NSF-ITR-IDM Award #0325464 and #071444.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

The sheer mass of available documents on the current Web and the insufficient representation of knowledge contained in documents make it quite burdensome for humans to find the right documents. A major shortcoming of the current Web is that information is targeted towards human and as such a human always has to be in the loop to interpret the information given in documents. In 2001, Berners-Lee, Hendler and Lassila [5] presented a vision of the Web called the Semantic Web in which Web content will be given well defined semantics, thereby making it more machine processable, empowering machines to act more on behalf of humans. The Semantic Web is about developing technologies that will enable machines to make more sense of the Web, with the result of making the Web more useful for humans. The World Wide Web Consortium (W3C) has put forward a layered architecture for the Semantic Web that shows the hierarchy of languages where each layer exploits and uses the capabilities of the layers below. At the lowest level are Unicode and Uniform Resource Identifier (URI). Unicode is a standard that allows for a consistent representation and manipulation of textual data expressed in most of the world's languages. URI is a compact string of characters used to identify or name a resource on the Web. On top of this layer is the Extensible Markup Language (XML) which allows for syntactic interoperability. The semantic layers then begin with the Resource Description Framework (RDF) [31] which sits on top of XML. The RDF schema (RDFS) provides basic vocabulary for RDF. On top of RDF sits the Web Ontology Language (OWL). OWL extends RDFS by making it possible to express complex relationships between different RDFS classes as well as to express more precise constraints on specific classes and properties. The rest of the layers contain technologies that are not yet standardized or currently undergoing standardization efforts such as the Rule Interchange Format (RIF) or just ideas of what should be implemented to realize the Semantic Web. In this thesis, we focus on RDF, since it is the building block of the semantic layers of the Semantic Web. However, our work can also be applied to OWL.

RDF provides a simple data model for describing entities in the Semantic Web in terms of named relationships and their values. The central notion of RDF is that of a resource which can be any entity that is uniquely identified by an IRI (Internationalized Resource Identifier) in the Semantic Web. IRIs are a general form of URIs that can be used to identify any entity. The fundamental construct in RDF is a statement. RDF allows for making statements about how resources in the Semantic Web are related, in the form of triples.

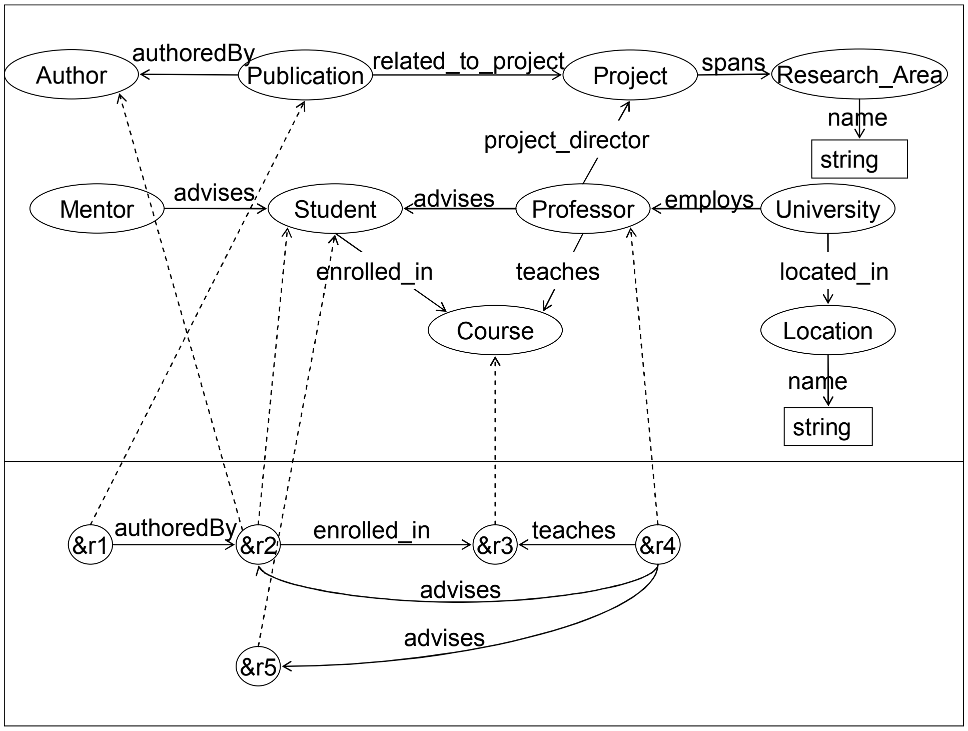
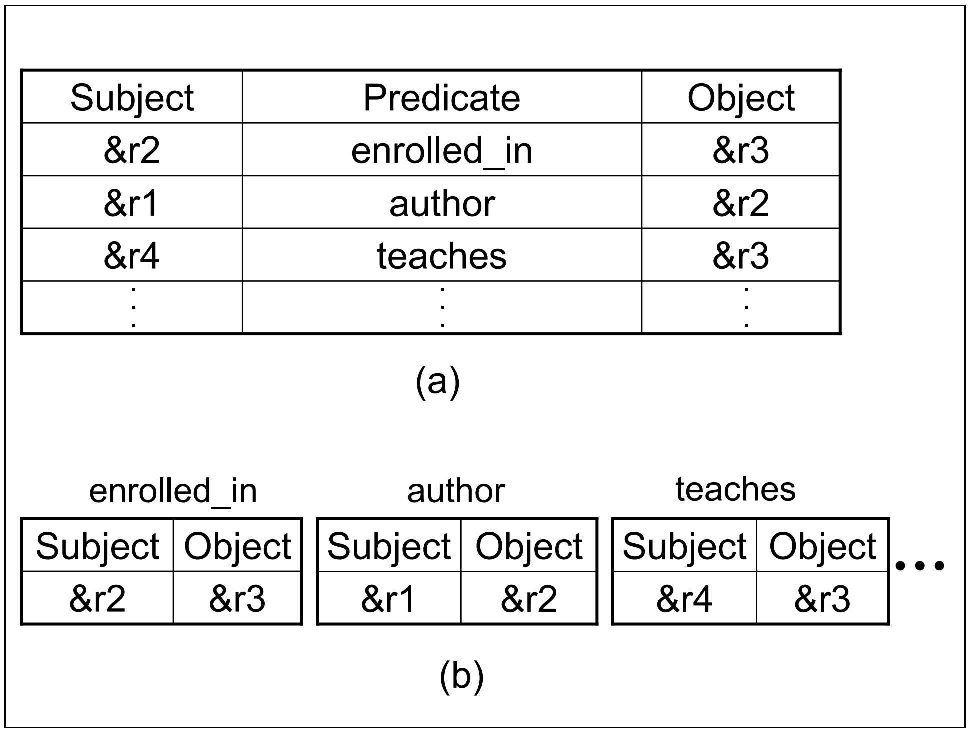


Figure 1.1: RDF Schema and Instance Graphs

A statement (subject, property, object) asserts that a resource which is denoted as the subject has a property whose value is the object (the object may be another resource or a literal). For example, suppose we want to make an assertion that states that a resource (in this case a person) with unique identifier http://www.example.edu/authors/author1 is the author of another resource (in this case a publication) with unique identifier http://www.example.edu/publications/publication1. We assert this by the RDF statement (ex:-publications/publication1, exterms:authoredBy, ex:authors/author1) where the prefixes ex and exterm are aliases for the namespaces http://www.example.edu and http://www.example.edu/terms (for the terms that are used by an example university). RDF statements can be represented graphically where labeled nodes represent the subject and object (labeled with their respective unique identifiers) while a labeled edge from the subject to the object, labeled by the value of the property, represents the property as shown in the bottom part of figure 1.1. Similarly, RDF's companion specification RDFS [7] provides special vocabulary for describing domain vocabularies. Domain vocabularies describe the types of entities, i.e., classes (e.g., Author) and types of relations or properties (e.g., authoredBy) in the domain, as shown in the top part of figure 1.1. RDFS also provides a special vocabulary of metaclasses and metaproperties for describing domain vocabularies. The metaclass rdfs:Class/rdfs:Property defines instances of Classes/Properties. Properties are further defined in terms of the classes whose instances they may apply to (called their domain) and those whose instances they may take as values (called their range). RDFS allows both classes and properties to be organized into subclassOf/subpropertyOf hierarchies. Thus, the definition of classes/properties may also contain information about which classes/properties they specialize using the rdfs:subclassOf/rdfs:subpropertyOf properties. Resources are also classified based on the classes they belong to, i.e., resource typing, using the same model and the rdf:type property.

1.1. Managing and Querying RDF

The growing interest in RDF with its accompanying schema language RDFS as a metadata and semantic data representation standard in the Semantic Web is spurring on the development of large-scale storage and high-level querying systems for RDF data. As observed at the 2008 Semantic Technology conference, a number of commercial applications such as those from Talis and Twine already use and apply very large RDF datasets, and continue to seek techniques to manage ever increasingly larger datasets. A variety of systems have been proposed for storing RDF ranging from main memory systems such as Jena [11][55], BRAHMS [28], to disk-based systems such as Sesame [9], Jena [11], YARS [21], RSSDB [30], Oracle RDF [61]. In order to exploit the maturity and wealth of research that has been invested in Database Management Systems (DBMS), some of the proposed systems for storing and querying RDF such as Sesame [9], Jena [11], RSSDB [30], Oracle RDF [61] employ a Relational/Object Relational Database backend (also see [50] for a survey) that shred the RDF graph into relations. The two most common techniques of shredding RDF graphs into relations are the schema-oblivious technique and the schema-aware technique. The schema oblivious technique stores all triples in one relation as shown in figure 1.2a, while the schema-aware technique has one relation for each property type as shown in figure 1.2b (see [51] for a taxonomy). Other storage schemes augment the triple store with an additional column such as context in YARS [21] or model id in Oracle [61]. Native stores such as BRAHMS [28], designed specifically for the needs of the RDF data model have also been proposed. Typically, a high level query language for querying RDF [29][32][42][43] has also been proposed along with each of the proposals for storing RDF. Many of the proposed query languages for querying RDF, support graph pattern matching as the primary query paradigm. In this paradigm, a query is a definition of a graph pattern that is to be matched in the database and the result of the query is the list of all occurrences or *matchings* of the graph pattern. Query processing techniques vary depending on the underlying storage techniques. For example, for systems that are built on relational database systems, queries are translated from an RDF query language such as SPARQL to SQL and then pushed to the underlying database for processing. One implication of the graph shredding storage approach with respect to query processing is that reconstructing subgraphs from database relations is done using join operations.



Subject	Predicate	Object
&r2	enrolled_in	&r3
&r1	author	&r2
&r4	teaches	&r3
⋮	⋮	⋮

(a)

enrolled_in

Subject	Object
&r2	&r3

author

Subject	Object
&r1	&r2

teaches

Subject	Object
&r4	&r3

•••

(b)

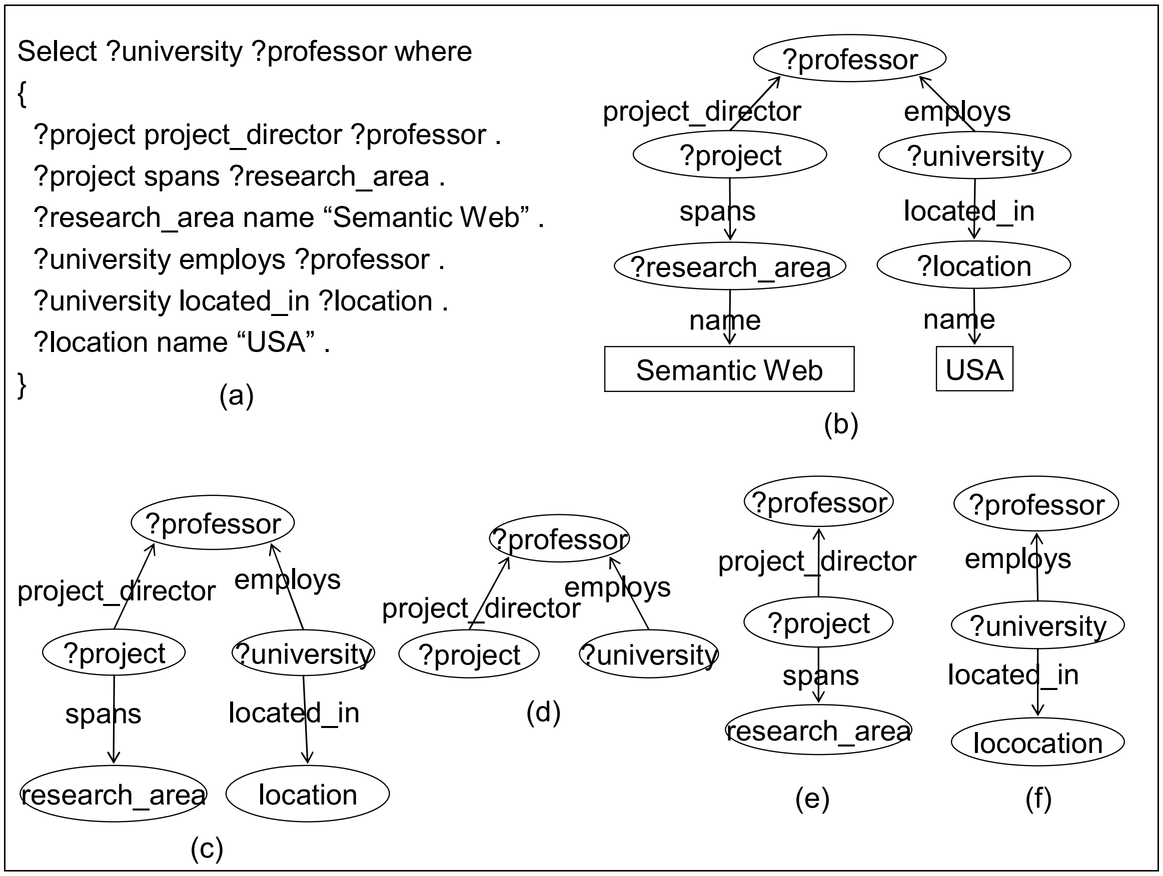
Figure 1.2: RDF Storage Mechanisms (a) Schema Oblivious Mechanism and (b) Schema aware Mechanism

This joining of triple patterns is often an expensive operation, necessitating an optimization step that helps reduce the query processing cost as much as possible. It is often the case that queries on RDF data will require several joins and unfortunately, these will often not be the primary key

– foreign key variety of joins. Therefore, optimizing the order of joins for queries on RDF data stores is arguably more important than it is for traditional relational databases.

1.2. RDF Graph Pattern Query Optimization

Independent of the technique in which the joins of the triple patterns is achieved in the different systems, the order in which the joins are performed is a crucial optimization step. Thus, the query optimizer needs to compare several alternative ways of computing the joins to determine the best way. Each way of executing the join is referred to as a query plan where a query plan is typically a tree of operators (including other operators besides the join operator). Each of the alternative query plans is associated with a processing cost so that the goal of the optimizer is to find the plan with the least cost which is referred to as an optimal plan. One of the key parameters the query optimizer needs to determine an optimal plan for executing the query is the size of the intermediate results. The estimation of the cardinality of intermediate join results is equivalent to estimating the frequency of sub-patterns of a query graph pattern. These cardinalities are estimated using a statistical summary of the data maintained by the optimizer. In many systems the exact nature of the statistical summary differs. In relational databases, the traditional approach is to keep the size of each relations and the number of distinct attribute values for each attribute of each relation. Further, histograms may be used to keep the distribution of values for each attribute in each relation. For the XML data model, several different summaries have been proposed (such as bloom histograms, XSketch synopses, correlated subpath tree, StatiX, etc), in the literature all of which exploit the tree nature of XML data as such, it is unclear how these techniques can handle graph structured data such as RDF. This thesis focuses on building statistical summaries of graph structured data needed for optimizing graph pattern queries and efficient techniques for estimating the frequency of patterns in a graph database from the statistical summaries. We note that a graph pattern query may contain constraints in the sense that the components of the triples it contains may be bound to a particular value. In this work, we focus on graph pattern queries which contain triple patterns which may have constraints on the subjects and/or objects. As an example, figure 1.3a below shows a SPARQL [42] query (namespaces are omitted for brevity) that asks for professors employed in universities in the United States who direct a Semantic Web project. The corresponding graph pattern for this query is shown in figure 1.3b. In this case, two out of the six triple patterns of the graph pattern have the objects bound to literal values while. However, suppose we first evaluate the two "name" constraints on research area and location, then as shown in figure1.3c, three joins will be needed to process this query.



```
Select ?university ?professor where
{
  ?project project_director ?professor .
  ?project spans ?research_area .
  ?research_area name "Semantic Web" .
  ?university employs ?professor .
  ?university located_in ?location .
  ?location name "USA" .
}
```

Figure 1.3: (a) A SPARQL Query Involving Several Join Operations (b) Its Corresponding Graph Pattern and (c,d,e,f) Sub-graph Patterns

One possible join order is to first join the necessary triples of property type “spans” to those of “project_director”, then joining the result from the first operation to the necessary triples of property type “employs” and then finally joining the result from the second operation to the necessary triples of property type “located_in”. However, to optimize this query, accurate estimates of the intermediate patterns leading up to the actual graph pattern of the query, such as those shown in figures 1.3d, e and f, are needed.

Noting that (1) the number of possible subgraphs in a graph database could be exponential and (2) it is more expedient if the estimates are computed without disk accesses, since they are needed at optimization time, we focus on techniques that summarize subgraphs in the graph database so as to fit in the available memory budget. Obviously, such a summary will be useful only if it captures the interactions among subgraphs. Our choice of subgraphs for capturing interactions amongst graphs is strengthened by the observations in [52], [59] and [60], where it is shown that subtrees and subgraphs perform better than paths in capturing interactions among graphs and trees, respectively. However, the number of unique subgraphs greatly exceeds the number of paths in a graph. It is thus infeasible to examine all subgraphs so we consider examining only subgraphs of size at most maxL. If the number of subgraphs is still too large, efficient pruning techniques will be needed. We propose two summaries that differ in their representation of subgraphs and pruning techniques: the Pattern Tree (P-Tree) and the Maximal Dependence Tree (MD-Tree).

Given a list of patterns and their frequencies, a simple way of representing the patterns is through the use of a hash table. However, we can more concisely maintain this information using our P-

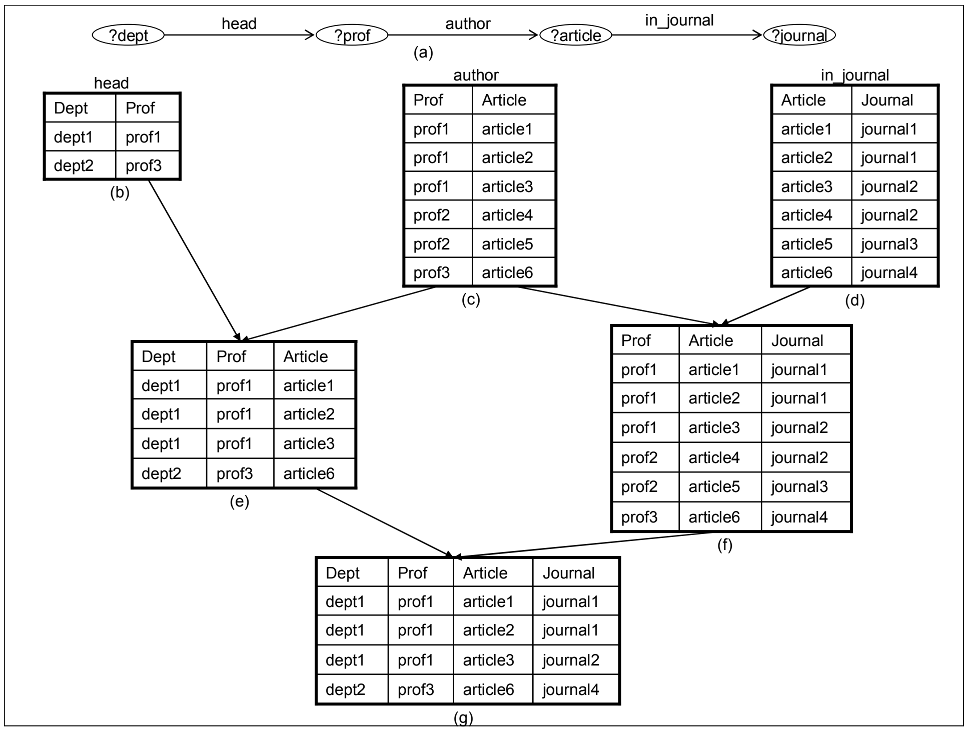
Tree technique which organizes the patterns in a prefix tree to save space needed for storing the patterns. The pruning technique of the P-Tree is based on two insights: (1) the frequency of a graph may be close to that of a function of its subgraph; and (2) information about the importance of subgraphs could lead to characterizing some as more important than others. For example, frequent subgraphs from a query workload are more important than infrequent ones for tuning purposes. We prune the P-Tree by blending the significance of patterns for estimation and for tuning purposes.

The MD-Tree is based on the observation that edge types in certain positions in patterns may largely determine the frequencies of the patterns. The idea then, is to assess the edge position with the greatest influence on the frequencies of the patterns. If no position of great influence exists, we assume that edges occur independently at each position. The pruning technique of the MD-Tree is based on the observation that high-order statistical dependencies often exist among subgraphs. It may be prohibitive to keep all such dependencies; thus, we attempt to capture the most informative dependencies in the given space.

To estimate the frequency of a pattern that is larger than maxL, we systematically divide it into patterns of length maxL and maxL-1 and combine the frequencies of these patterns to obtain an estimate for the frequency of the large pattern. Given a pattern p of length k, we divide p into k-maxL+1 patterns of length maxL where each succeeding pattern of length maxL overlaps the preceding one in all but one edge. We then estimate p as follows:

$$freq(p_1) * \frac{freq(p_2)}{freq(p'_2)} * ... * \frac{freq(p_{k-\max L+1})}{freq(p'_{k-\max L+1})}$$

where $freq(p_i)$'s are the frequencies of the patterns of length maxL and $freq(p_i')$'s are the frequencies of the overlapping patterns of length maxL-1. This is illustrated in the following example.



Dept	Prof
dept1	prof1
dept2	prof3

Prof	Article
prof1	article1
prof1	article2
prof1	article3
prof2	article4
prof2	article5
prof3	article6

Article	Journal
article1	journal1
article2	journal1
article3	journal2
article4	journal2
article5	journal3
article6	journal4

Dept	Prof	Article
dept1	prof1	article1
dept1	prof1	article2
dept1	prof1	article3
dept2	prof3	article6

Prof	Article	Journal
prof1	article1	journal1
prof1	article2	journal1
prof1	article3	journal2
prof2	article4	journal2
prof2	article5	journal3
prof3	article6	journal4

Dept	Prof	Article	Journal
dept1	prof1	article1	journal1
dept1	prof1	article2	journal1
dept1	prof1	article3	journal2
dept2	prof3	article6	journal4

Figure 1.4: Estimating the cardinality of Large Patterns

Given the graph pattern query shown in figure 1.4a and suppose the database contains the tables shown in figures 1.4b, c and d. Suppose also that our summaries contain patterns of length at most 2, then our summaries will have the cardinality 4 for the join of “head” and “author” and 6 for the join of “author” and “in_journal” as shown in figures 1.4e and f. Since the query pattern is of length 3, we estimate the cardinality of the result of the query using the cardinalities of the patterns of length 2 and 1 in the summaries by dividing the query pattern into overlapping patterns of length 2 where the overlap is in all but one edge. In this case, the overlap is in one edge. Then we take the products of the cardinalities of the patterns of length two and then divide by the cardinalities of the overlap. In this case we will estimate the result of the query as (4 * 6)

/6 since the overlap is the “author” edge which has a cardinality of 6. In this example, our estimate is the exact cardinality of the result as shown in figure 1.4g.

In particular, this work makes the following contributions:

1. *Generic Representation for an RDF Graph Pattern*. With the multiple classification of resources allowed in the RDF data model, graph patterns may exist in the instance graph that are not explicitly represented in the schema graph. We create a *Semantic and Structural Summary* for RDF/S by adding structural information from the RDF instance graph to a Semantic Summary [3] created from the RDF schema.
2. *Graph Sequencialization.* Using the *Semantic and Structural Summary*, we adapt a technique described in [58] for creating a partial ordering of edges in an undirected graph to obtain a variable length sequence for RDF graph patterns.
3. *Summarization Framework.* Given that the number of unique graph patterns may be exponential, we develop heuristics for pruning patterns of at most size maxL to fit a given memory budget.
4. *Estimation Framework.* Using the proposed summaries, we show how to obtain estimates of the cardinality of both patterns that are represented in the summary and those that are not explicitly represented in the summary. The estimates obtained help guide a query optimizer in choosing an optimal query plan.
5. *Experimental Validation.* We conduct experiments on real-life and synthetic datasets which validate our approach. Our experiments show the efficiency of the proposed summaries in providing estimates that help in optimizing both unconstrained join queries and constrained join queries over RDF graphs

2. Related Work

2.1. Optimizing RDF Graph Pattern Queries

An optimization approach for RDF Graph patterns that is based on the selectivity of RDF triples was proposed in [49]. To determine the selectivity of resources, they simply adapted the techniques developed for Relational Databases to RDF. They first define a selectivity measure for resources, based on this selectivity for resources, they define the selectivity for a triple pattern and then a graph pattern. Similar to our approach, they determine the selectivity of graph patterns of size 2 (i.e., two joined triple patterns) by enumerating and storing all possible graph patterns of size 2 and their frequencies. However, unlike in our approach, they only consider joined patterns of size 2 that are obtainable from the schema and as such, may miss some patterns. Furthermore, they do not consider approaches for pruning the summary such that it fits a given memory budget as well as techniques for tuning the summary to favor certain patterns.

The optimization unit of an implementation of the SPARQLeR query language [32], adopts a greedy approach for optimizing supported graph pattern queries. Given the graph pattern for a query such that edges in the graph pattern are annotated with the cardinality of triples that satisfy the triple pattern it represents, a query plan is constructed by beginning with the edge which has the least cardinality and examining its adjacent edges for the edge with the next smallest cardinality, using an algorithm similar to that used for growing a minimum spanning tree. Each time a new edge is added to the plan, its adjacent edges are added to the pool of adjacent edges from which the next edge with the smallest cardinality is chosen. This process is repeated until all edges of the graph have been visited. In contrast to our approach, this does not provide for nor utilize the estimates of the cardinality of larger patterns in constructing the query plan. As such, this technique may be less effective in finding optimal plans.

RDF Systems such as [11][30][61] which utilize a Relational/Object Relational Database backend translates the query in the supported RDF query language into SQL which is then pushed down to the underlying Database backend for processing. As such, the task of query optimization is essentially pushed down to the optimizer of the Database backend.

2.2. Statistical Summaries for Cardinality Estimation

The problem of summarizing data for the purposes of estimating the cardinality of join results has been studied for the Relational data model and can be largely categorized into 1) those which summarize the attribute values and their correlations for attributes within a single relation such as the one-dimensional and multi-dimensional histogram-based techniques of [26][27] and [41], respectively, and the Statistical Interaction Models proposed in [15], 2) those which attempt to capture the join dependencies across attribute values of attributes in multiple relations such as the Probability Relational Model proposed in [18].

In the former case, the correlations of attribute values maintained are mainly targeted at estimating the cardinality of tuples in a single relation which satisfy some predicate. The cardinality of joins is estimated by combining the statistics maintained for the join attributes in their respective relations with the assumption that the values of the join attributes are uniformly distributed across the tables. The typical statistics maintained in this approach is the size of each relations and the number of distinct attribute values for each attribute of each relation. Further, histograms may be used to keep the distribution of values for each attribute in each relation. This approach proves inadequate in capturing the join dependencies that exist amongst RDF triples. To capture the joint frequency distribution of multiple attributes across multiple relations, the use of Probability Relational Models (PRMs) was proposed in [18]. PRMs rely on a Bayesian network that exploits conditional independence to approximate the joint frequency distribution of attributes of tuples across tables, joined via a foreign key. Since this approach only considers foreign-key to primary-key joins, it proves inadequate for capturing many-to-many dependencies that exist amongst attribute values of tables, which is typically the case in semi-structured data. Further, if RDF triples are stored in relations using a schema-aware technique, then both attributes in all relations form the key. In such a scenario, this technique does not clearly specify how the join dependencies can be achieved. If however, a schema-oblivious technique is used, it becomes even more unclear how the join dependencies can be achieved since all attributes form the key of the single relation. The Tuple Graph Synopses (TUGS) was proposed in [47] as a graph-based summary for cardinality estimation for relational databases. Given a schema graph which defines the join relationships amongst relations, this technique constructs a graph of the data contained in all relations and their join dependencies with respect to the schema graph. The TUGS synopsis for the data is then constructed by a systematic summarization of the instance graph. However, it is unclear how this approach may be applied to RDF graph stored using the schema-oblivious approach, nor RDF graph patterns which require self-joins, since the TUGS model does not consider self-joins of a relation. A technique that seamlessly copes with RDF patterns, regardless of the storage model used to map the data into relations is desirable.

The Jena property Table approach [55] proposes a technique that utilizes certain frequently occurring patterns in RDF data for creating a Relational schema for RDF in such a way that query patterns which would otherwise have been translated into join operations now amount to select operations on a single table. In effect, they pre-compute and store the joins of the frequently occurring patterns so that this approach attempts to optimize join operations over RDF data by avoiding the joins at run time where possible. However, the usefulness of this approach is limited to the particular frequently occurring patterns stored in a relational table. The join operations can not be avoided for arbitrary patterns.

The data summarization and result cardinality estimation problem has gained significant research interest for the XML data model for simple path expressions [1][12][34][36][40][54][57][52], branching path expressions [12][40][57] and twig patterns [12][39][57][52]. Although all these efforts propose ingenuous summarization and estimation framework for XML, most of these techniques assume that the XML data is modeled as a tree, so that they are unsuitable for RDF data which is modeled as a directed labeled graph. More importantly, all these techniques are targeted at queries which exhibit tree patterns so that it is unclear how they apply to arbitrary graph-structured queries. Although the proposals of [40][52] assume graph structured XML data (considering idrefs), the queries are still assumed to be either simple path expressions or twigs, which are modeled as trees, so that it is unclear how queries with a more complex structure can be handled using these techniques. The estimation value which we defined for pruning patterns in the P-Tree is similar in spirit to the notion of δ-derivable twigs introduced in [52] for pruning twigs whose estimated frequencies are within δ error of their true frequencies. However, using the pruning technique of [52] may cause the pruning of a pattern that, if not pruned, may have resulted in an even larger reduction of the size of the summary by causing the pruning of even more patterns. In contrast, our value-based approach makes a more informed choice of patterns to be pruned.

Research in Data Mining such as in [6][37] have investigated the problem of computing a condensed frequent pattern base for estimating the cardinality of frequently occurring patterns in a large item base. However, the construction technique of the pattern bases used for this purpose does not cope with the challenging aspect of fitting a size budget. Further in contrast to our work, these efforts focus only on estimating the cardinality of frequently occurring patterns.

2.3. Graph Indexing

Our work is also related to research in the area of graph indexing, which indexes fragments of graphs in a database consisting of a collection of many disconnected graphs, for optimizing graph containment queries. In this setting, given a query graph, all graphs which contain the query graph are returned as the result of the query. A graph containment query is processed in two steps. The first step retrieves a candidate set of graphs that contain the indexed fragments of the query graph. The second step uses subgraph isomorphism to validate each candidate graph. Several efforts have been made in using graph-indexing schemes to reduce the cost of processing graph containment queries. GraphGrep [46] uses a path-based indexing approach that selects all paths of up to length l_p as the indexing feature. The size of the candidate set obtained in the first step could be large, since path fragments do not maintain the overall structure of graphs. Since the size of the candidate set is large, many isomorphism tests have to be performed for validating the graphs it contains. To cope with this, GIndex [59] uses frequent graph fragments as the indexing feature. To reduce the large (potentially exponential) number of frequent fragments, only discriminative frequent fragments are kept. Although GIndex performs the index construction as a pre-processing step, the index construction time may be large since it uses a mining approach which requires performing graph and subgraph isomorphisms, for discovering the discriminative frequent fragment. Noting that the set of frequent graph fragments contain many more tree than non-tree structures, Tree+Δ [60] indexes frequent trees, which it discovers using a frequent tree mining algorithm that avoids the expensive graph and subgraph isomorphisms, thereby reducing the large index construction time of GIndex due to graph mining. On demand, Tree+Δ further reduces the size of the candidate set by selecting a small portion of discriminative non-tree features related to query graphs only. As a complement to these efforts, our work allows for optimizing the subgraph isomorphism tests, in the second step, using estimates of the cardinalities of both indexed and non-indexed fragments of the query. In addition, our technique can also be applied to a large connected graph.

The rest of this work is as follows. In section 3, we give the preliminaries and background needed for our work. In particular, we formally discuss our notion of the RDF data model (RDF Schema and RDF instance graphs). We also formulate the problem addressed in this work. In section 4, we discuss the construction, pruning and estimation algorithms for our statistical summaries, the Pattern Tree (P-Tree) summary and the Maximal Dependence Tree (MD-Tree) Summary. In section 5, we show the results of experiments we conducted to evaluate the performance of our summaries in terms of the accuracy of the estimates obtained from them and the effectiveness of these estimates for query optimization. In section 6, we conclude our work and give directions for future work.

3. Preliminaries and Background

3.1. RDF Data Model

Let C, P, I and L be the sets of class Universal Resource Identifiers (URIs), property URIs, instance URIs and literal values, respectively; the concepts of RDF schemas and instances can be formalized as follows.

Definition 1. RDF Schema Graph: An RDF schema graph $G_S = (V_S, E_S, \lambda_S, C, P)$ is a directed labeled graph where V_S is the set of nodes of G_S and E_S is the set of edges (i.e. subset of cartesion product of V_S) of G_S and λ_S: $(V_S \cup E_S) \rightarrow C \cup P$ is a surjective labeling function that maps vertices and edges of G_S to class and property URIs, respectively, such that $\lambda_S(v) \in C$ and $\lambda_S(e) \in P$ for any $v \in V_S$ and $e \in E_S$

Definition 2. RDF Instance Graph: An RDF instance graph $G_I = (V_I, E_I, \lambda_I, \tau, I, P, L)$ defined on a schema graph G_S is a directed labeled graph where V_I and E_I are sets of vertices and edges of G_I, respectively; $\lambda_I : (V_I \cup E_I) \rightarrow I \cup L \cup P$ is a surjective function that maps vertices and edges of G_I to instance URIs or literals or property URIs, respectively such that $\lambda_I(v) \in I \cup L$ for any $v \in V_I$ and $\lambda_I(e) \in P$ for any $e \in E_I$. On the other hand, $\tau : V_I \rightarrow 2^{V_s}$ is a function which maps nodes of G_I to sets of nodes of G_S. This typing of nodes is such that for any edge $e = (u, v)$ in G_I, if $\lambda_I(e) = p$, then there is an edge $e' = (u', v')$ in E_S for which $\lambda_s(e') = p$, $\tau(u) = \{u'\}$ and $\tau(v) = \{v'\}$.

Note that our model of the instance graph consists of only ground RDF graphs which do not contain reified statements (i.e. assertions about statements).

Definition 3. RDF Graph Pattern: An RDF graph pattern or simply a pattern $G_P = (V_P, E_P, \lambda_P, P)$ is a connected edge-labeled directed graph where $V_P \subseteq N$ and $E_P \subseteq E_S$ and $\lambda_P: E_P \rightarrow P$ and for $e \in E_P$ $\lambda_P(e) = \lambda_S(e)$.

Problem Formulation. This work focuses on exploiting information in an RDF graph to improve the statistics needed for cost-based query optimization of both unconstrained and constrained join queries posed over RDF data stores. In particular, we focus on improving the statistics needed for estimating the cardinality of equi-join operations. This amounts to obtaining information about the frequencies of patterns in an RDF graph.

Our approach is to model a query as a graph pattern defined by edge labels and topology (i.e., the nodes are considered variables and thus are not bound to any particular label).

Definition 4. Given an instance graph G_I and a query pattern G_P, we say that G_P has a *matching* in G_I if we can map every edge in G_P to an edge in G_I with the same label i.e., for $e \in E_P$, $\lambda_P(e) = \lambda_I(e')$ where $e' \in E_I$. This matching is such that the topology of the nodes and edges in G_P are preserved. Thus, a matching of a query pattern in an instance graph constitutes a result of the query.

G_{I1} and G_{I2} are two unique subgraphs in G_I matching G_P, if at least one node in G_{I1} is different (labeled with a different URI) from its corresponding node in G_{I2}. From the foregoing, it is clear that all unique matchings of a query pattern constitute the result of the query. Thus, our goal is to obtain the frequency or cardinality of all unique matchings of a particular pattern.

Definition 5. The *frequency* of a query pattern is the number of matchings it has in the instance graph.

Unfortunately, it is prohibitive to count all matchings of a pattern from G_I at query optimization time so that the use of a summary of G_I is necessary. Thus, the problem we address in this work can be stated more precisely as follows:

Given an RDF schema graph and a corresponding instance graph, and a memory budget B, create a summary of size at most B, from which we can obtain accurate estimates of the number of unique matchings of a pattern from the instance graph.

3.2. Canonical Labeling of Graphs

In order to count the frequencies of patterns, we need an efficient way to uniquely enumerate the patterns. In other words, we need a canonical label for patterns. In this section, we discuss the DFS Coding canonical labeling and the gSpan algorithm which we adopt in this work for labeling and counting the frequencies of matchings of a pattern.

With minimal modifications, efficient pattern mining algorithms, such as gSpan [58] and [53] (which uses disk-based indexes for limited memory settings) can be used to discover subgraphs matching patterns and count their frequencies. In general, these techniques first develop a canonical label (i.e., unique code) for a graph; then subgraph frequencies are computed based on the canonical label. In the next paragraph, we briefly review the *minimum Depth First Search (DFS) code* [58] of a graph as its canonical label that we adopt in this work.

3.2.1. DFS Coding

This technique uses a Depth First Search (DFS) traversal of a graph to transform it into an edge sequence called DFS code. Each edge (u, v) in the graph is represented by a 5-tuple $<i, j, l_i, l_{(i, j)}, l_j>$, where i and j are integers denoting the DFS discovery times of nodes u and v and l_i, l_j and $l_{(i, j)}$ are the labels of u, v, and the edge (u,v) respectively. The edges are then ordered by listing those in the DFS tree (tree edges) in the order in which they were discovered, then inserting the remaining edges into the ordered list as follows: Given a tree edge (u, v), all non-tree edges from v are listed immediately after (u, v); if (u_i, v_j) and (u_i, v_k) are two such non-tree edges, (u_i, v_j) is listed before (u_i, v_k) only if $j < k$. Since a graph can have multiple DFS codes, the minimum, obtained based on a linear ordering of all its DFS codes is chosen as its canonical label.

3.2.2. gSpan Algorithm

Given an undirected graph G and an integer F, the gSpan algorithm allows for enumerating all subgraphs of G whose frequencies are at least F, where the minimal DFS code of each subgraph is its canonical label. The algorithm iteratively generates and counts all unique subgraphs of length i+1 from those of length i, whose frequencies are at least F (the length of a subgraph is defined as the number of edges it contains). Each generated subgraph of length i+1 with frequencies less than F is pruned on the assumption that it cannot lead to the generation of any new frequent subgraph (i.e., with frequency at least F). Details of the algorithm can be found in [58].

4. Summarization and Estimation Framework

In this section, we discuss our summarization and estimation framework for graph patterns. We begin by first discussing the Semantic and Structural Summary, then we discuss the P-Tree summary and how to obtain estimates of patterns from it. Next we discuss the MD-Tree summary and how to obtain estimates of patterns from the MD-Tree, then we discuss how to obtain estimates for patterns with length greater than maxL.

4.1. Semantic and Structural Summary

Recall that at least a node URI distinguishes two unique matchings of a query pattern in the instance graph so that to summarize the matchings of patterns in the instance graph, we need a general representation that encompasses all unique matchings of any given query pattern. We achieve this by refining the RDF Schema graph to include all patterns that exist and may exist in the RDF instance graph and derive our representation from the refined schema graph. In a previous work [3], we introduced the RDF *Semantic Summary* as a data structure that captures all possible paths which can be obtained from the RDF schema based on either explicit definitions in or deductions from the schema.

Definition 6. Given an RDF Schema graph $G_S = (V_S, E_S, \lambda_S, C, P)$, a Semantic Summary $G_{SS} = (V_{SS}, E_{SS}, \lambda_{SS})$ is a directed labeled graph where E_{SS} the set of edges of the graph is given by

$E_{SS} = \{e_i \mid e_i \in E_S, \lambda_S(e_i) \neq \lambda_S(e_j), i \neq j\}$.

V_{SS} the set of nodes of the graph is given by

$V_{SS} = \{\text{domain}(u) \cup \text{domain}(v) \mid u \in E_S, v \in E_S, \lambda_S(u) = \lambda_S(v)\} \cup \{\text{range}(u) \cup \text{range}(v) \mid u \in E_S, v \in E_S, \lambda_S(u) = \lambda_S(v)\}$ where the functions domain/range gives the domain/range classes of an edge in E_S.

$\lambda_{SS}(e) = \lambda_S(e)$ for e in E_S

In other words, if multiple edges have the same label, these edges are merged into one by merging their respective source and destination nodes. If u and v are two nodes of G_S merged into w and if u′ and v′ are two nodes of the instance graph that are types of u and v, u′ and v′ are types of w in G_{SS}. The edges emergent from or incident on a node w in G_{SS} is the union of the respective edges emergent from or incident on all nodes in G_S that were merged into w.

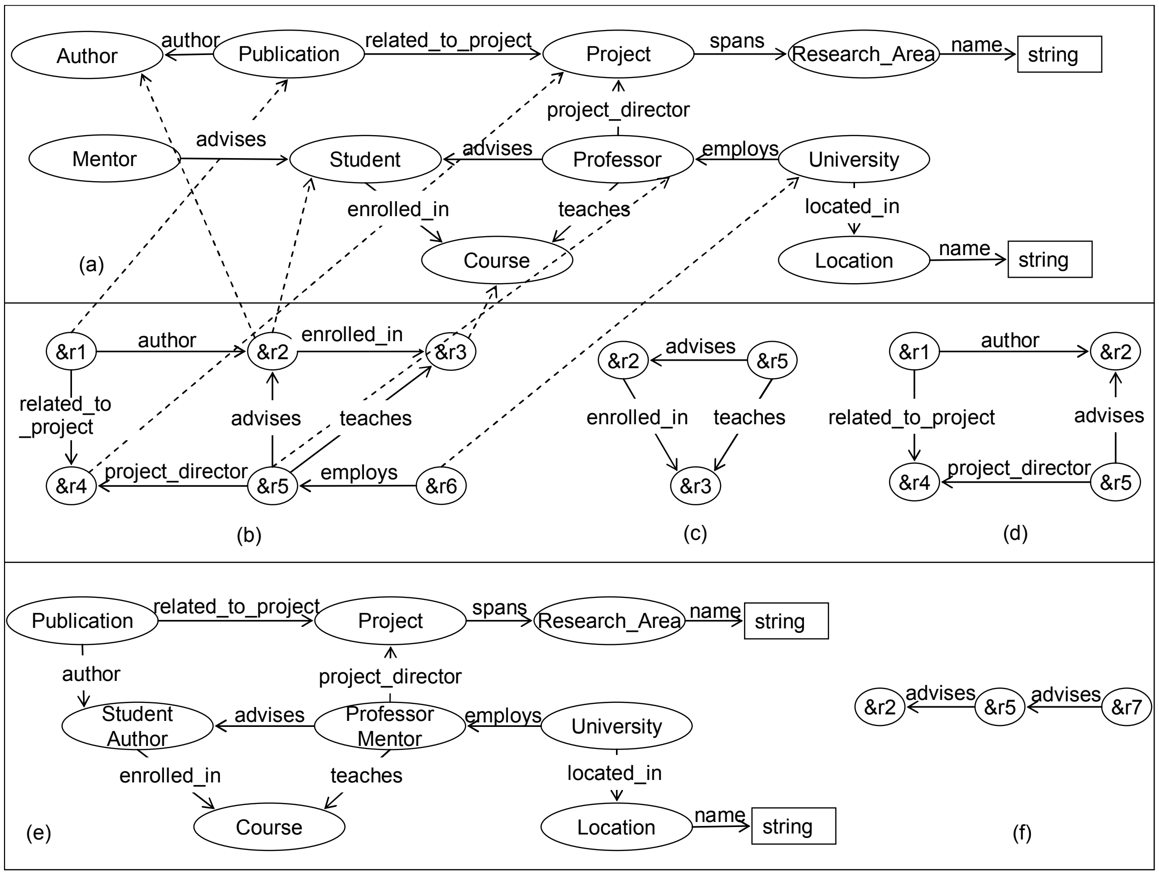


Figure 4.1: Sample RDF Schema Graph, Instance Graph, Graph Patterns and Semantic and Structural Summary.

Although G_{SS} captures all possible paths defined in the RDF schema, it may not describe the structure of all patterns in the instance graph. This stems from the multiple classifications of resources allowed in the RDF data model. We enhance G_{SS} with this capability by merging nodes in G_{SS} which have any instance nodes in common. The resultant nodes are then annotated with unique integer ids. Each edge in G_{SS} is also annotated with a unique integer id. We refer to this data structure as an RDF *Semantic and Structural Summary*. Essentially, a node in the Semantic and Structural Summary represents either a single node or a collection of nodes in the Semantic Summary. Example 1 illustrates the merging of nodes in the Semantic and Structural Summary to reflect all subgraphs in the instance graph

Example 1. *In the RDF schema graph shown in figure 4.1a above, the property type "advises" is defined on the domains "Mentor" and "Professor". A Semantic Summary of figure 4.1a represents these two classes by the same schema object with respect to the property "advises" as shown in figure 4.1e. Figure 4.1c and figure 4.1d show two subgraphs drawn from the instance graph shown in figure 4.1b. While the former subgraph is reflected in the schema, the latter is not because the resource &r2 is multiply classified as an "Author" and a "Student". A Semantic and Structural Summary of figure 4.1a will incorporate this subgraph by representing these two classes with the same schema object as shown in figure 4.1e.*

Formally, if *extent(u)* is a function that gives the nodes in the instance graph G_I which are mapped to a node u in G_{SS}, we define the RDF Semantic and Structural Summary as follows.

Definition 7. RDF Semantic and Structural Summary: Given an RDF Semantic Summary $G_{SS} = (V_{SS}, E_{SS}, \lambda_{SS})$, an RDF Semantic and Structural Summary is a directed labelled graph $G_{SSS} = (V_{SSS}, E_{SSS}, \alpha, \lambda_{SSS})$ where $\lambda_{SSS} : E_{SSS} \rightarrow P$ is a labelling function that maps elements of E_{SSS} to the set of property URIs, V_{SSS} the set of nodes of the graph is given by

$V_{SSS} = \{u \cup v \mid u \in V_{SS}, v \in V_{SS}, \text{extent}(u) \cap \text{extent}(v) \neq \varnothing\}$

E_{SSS} the set of edges of the graph is given by

$E_{SSS} = \{e = (u, v) \mid u \in V_{SSS}, v \in V_{SSS}, \exists e' = (u', v'), \lambda_{SSS}(e) = \lambda_{SS}(e'), u' \subseteq u, v' \subseteq v\}$

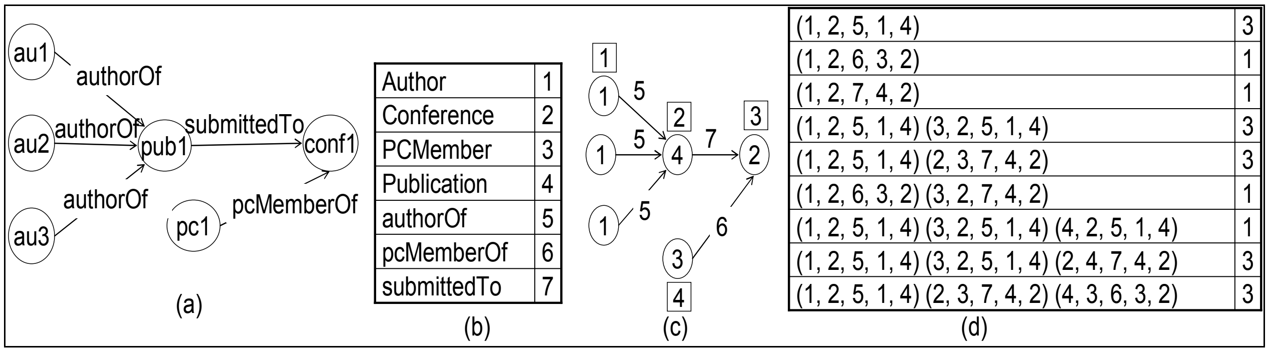
$\alpha : (V_{SSS} \cup E_{SSS}) \rightarrow N$ is a numbering function such that $\alpha : V_{SSS} \rightarrow N$ and $\alpha : E_{SSS} \rightarrow N$ are injections.

In other words, if two nodes of the Semantic Summary have a common member in the instance graph, the Semantic and Structural Summary merges the two nodes. Also every edge of the Semantic and Structural Summary is either an edge of the Semantic Summary or an edge resulting from the merging of nodes of the Semantic Summary.

The Semantic and Structural Summary differs from the structural summaries described in [1][19][40] in the sense that it may contain properties defined in the RDF schema which do not exist at all in the instance base. We allow the representation of such properties so that updates to the instance graph that eventually introduce them can be accommodated gracefully. Furthermore, the Semantic and Structural Summary may introduce spurious paths and cycles in the data summary. For example, if resource &r5 in figure 4.1b is also defined as an instance of the class *Author*, incorporating this in the Semantic and Structural Summary will cause the classes *Student*, *Author*, *Professor* and *Mentor* to be merged. The merged node will have the edge *advises* as a self loop so that the spurious pattern shown in figure 4.1f is seen as valid. We will investigate possible solutions to this problem in the future.

Having discussed the Semantic and Structural Summary, we now direct our attention to our proposed P-Tree and MD-Tree summaries. To create each of the proposed summary, we begin by generating all subgraphs of length at most maxL and counting their frequencies using a slight modification of the gSpan algorithm with input graph $G = (V, E, \lambda_I, \tau)$ and the frequency threshold set to one. However, we represent each edge $e = (u, v)$ in G by a 5-tuple $<i, j, \lambda_I(e), \tau(u), \tau(v)>$, where i and j are integers that denote the DFS discovery times of nodes u and v and λ_I and τ are the functions defined in section 3.1, with $\lambda_I(.)$ and $\tau(.)$ (i.e., the ranges of λ_I and τ) mapped to integers in accordance with the RDF Semantic and Structural Summary. The sequence of edges/quintuples obtained after the algorithm is run represents the structure of subgraphs of G; thus, given any two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ for which $v_1 = v_2$, it follows that $\tau(v_1) = \tau(v_2)$. While gSpan discusses the minimum DFS code in the context of undirected labeled graphs, for directed labeled graphs, we ignore edge directions during DFS traversal so as to maintain the connectivity of the graph, however the directions are kept implicitly in the quintuples. Example 2 explains how to obtain the minimal DFS code of a pattern.

Example 2. *Figure 4.2a shows a directed labeled graph that models conference information. All publication authors are classified as members of the class "Author", publications as "Publication" and so on. Concepts and edge labels are assigned integer ids as shown in figure 4.2c. To obtain the edge sequence for the subgraph au1 authorOf pub1, pub1 submittedTo conf1, pc1 pcMember of conf1, we begin DFS with the edge authorOf as it is lexicographically the smallest label. Traversal proceeds as shown by the boxed subscripts associated with the nodes to yield the pattern (1,2,5,1,4) (2,3,7,4,3) (4,3,6,3,2). Note that the direction of the edge labeled "pcMemberOf" is implicit in the sequence. Figure 4.2d shows all patterns of length at most 3 and their frequencies from the graph in figure 4.2c.*



Author	1
Conference	2
PCMember	3
Publication	4
authorOf	5
pcMemberOf	6
submittedTo	7

(1, 2, 5, 1, 4)	3
(1, 2, 6, 3, 2)	1
(1, 2, 7, 4, 2)	1
(1, 2, 5, 1, 4) (3, 2, 5, 1, 4)	3
(1, 2, 5, 1, 4) (2, 3, 7, 4, 2)	3
(1, 2, 6, 3, 2) (3, 2, 7, 4, 2)	1
(1, 2, 5, 1, 4) (3, 2, 5, 1, 4) (4, 2, 5, 1, 4)	1
(1, 2, 5, 1, 4) (3, 2, 5, 1, 4) (2, 4, 7, 4, 2)	3
(1, 2, 5, 1, 4) (2, 3, 7, 4, 2) (4, 3, 6, 3, 2)	3

Figure 4.2: Unique Edge Sequences for Subgraphs

Note that, in our directed graph model, an edge, for example (5, 1, 4) in figure 4.2, may appear in a pattern of length at least two, in one of three possible directions: forward, as in (1, 2, 5, 1, 4); backward, as in (3, 2, 5, 1, 4); or potentially in a self-loop, as in (2, 2, 5, 1, 4).

At this point, we have the necessary summary data to make accurate estimates for the sizes of joins (intermediate or final). If the number of joins is less than or equal to maxL, the estimate may be exact (and from empirical testing is always very close). Note, estimates are not always exact, since the gSpan algorithm provides an efficient way to determine frequency counts, they may differ slightly from a more straightforward, though very inefficient, technique of pre-computing all joins. When the number of joins is greater than maxL, formulas must be used to provide somewhat less accurate estimates

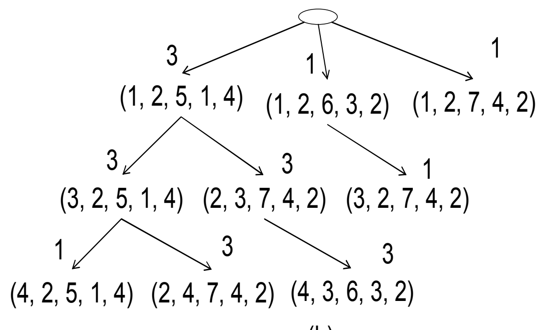
4.2. Pattern Tree (P-Tree)

Given the set of patterns and their frequencies, the P-Tree is a prefix tree representation of the patterns that achieves more compactness than the list of patterns. Its nodes are labeled with edge patterns (patterns of length 1) such that any pattern in *P* can be obtained by a concatenation of node labels on a path from the root. Also, each node is associated with the frequency of the pattern it represents.

(1, 2, 5, 1, 4)	3
(1, 2, 6, 3, 2)	1
(1, 2, 7, 4, 2)	1

(1, 2, 5, 1, 4) (3, 2, 5, 1, 4)	3
(1, 2, 5, 1, 4) (2, 3, 7, 4, 2)	3
(1, 2, 6, 3, 2) (3, 2, 7, 4, 2)	1
(1, 2, 5, 1, 4) (3, 2, 5, 1, 4) (4, 2, 5, 1, 4)	1
(1, 2, 5, 1, 4) (3, 2, 5, 1, 4) (2, 4, 7, 4, 2)	3
(1, 2, 5, 1, 4) (2, 3, 7, 4, 2) (4, 3, 6, 3, 2)	3

(a)



(b)

Figure 4.3: Pattern Tree

Definition 8. Given two patterns P_i and P_j with sequence(P_i) = $e_1, e_2, \ldots, e_k$ and sequence(P_j) = $e'_1, e'_2, \ldots, e'_m$ of lengths k and m respectively, we say that P_i is a sub-pattern of P_j and respectively P_j is a super-pattern of P_i if $m > k$ and $e_i = e'_i$ for $1 \le i \le k$. Further, we say that P_i is a maximal sub-pattern of P_j and respectively P_j is a minimal super-pattern of P_i if k equals m-1.

Lemma 1. Given a set *P* of patterns of length at most maxL, every pattern P in *P* of length m, $m \ge 2$ has exactly one unique maximal sub-pattern in *P*.

Proof. The canonical representation of patterns is a linearization of the patterns so that every pattern of length at least 2 has only one maximal sub-pattern.

By Lemma 1, pattern P_i is the parent of pattern P_j in the P-Tree, if P_i is its maximal sub-pattern. The root of the tree is an empty node whose children are patterns of length 1. We illustrate this with an example.

Example 3. *Figure 4.3b shows the P-Tree for the patterns in figure 4.3a, which are the same patterns introduced in figure 4.2a. Except for the root, each node is associated with an edge* *pattern and the frequency of the pattern it represents. Thus the frequency of the pattern (1,2,5,1,4) (3,2,5,1,4) (4,2,5,1,4) can be obtained by traversing the leftmost branch of the tree.*

Pruning the Pattern Tree.

To motivate the pruning of the P-Tree, we observe that given to two patterns P_i and P_j which have almost the same edge patterns, the frequency of P_i may be within δ of that of P_j, where δ is a small non-zero positive integer. The idea of our P-Tree approach then is to identify sets of patterns which have almost the same edge patterns such that for a set say *P*, the frequencies of patterns in *P* are within δ of at least one pattern in *P* say P_i. Thus, given P_i, the frequencies of all other patterns in *P* can be estimated within δ error so that we can safely eliminate all other patterns in *P* from the summary. In this section, we discuss how we construct a P-Tree that fits the given summary budget.

If the size of the P-Tree exceeds the budget, it has to be systematically pruned so as to avoid a large increase in its overall estimation error. The question then is; *which nodes are to be pruned and in what order?* To answer this question, we develop the concepts of *preference* and *estimation* values of patterns. We begin by introducing some notations.

Notations.

We denote a set of patterns of length k by $\boldsymbol{P}_{\mathbf{K}}$ and we define the function **freq** whose domain is the power set of patterns of length at most maxL and whose range is the set of positive integers, such that if X is a single pattern, freq(X) maps to the frequency of the pattern. If however, X contains more than one pattern, freq(X) maps to the sum of the frequencies of all patterns in X. Formally, we have that

$$\text{freq}(X) = \begin{cases} \text{frequency of p in } G_I, p \in X| & \text{if } |X| = 1 \\ \sum_{p \in X} \text{freq}(p) & \text{if } |X| > 1 \end{cases}$$

We define function **children** whose domain and range are the set of patterns of length at most maxL and the power set of patterns of length at most maxL, respectively, such that for a pattern P_i, children(P_i) maps to the set of children of P_i in the P-Tree.

To prune the P-Tree, we consider that in some scenarios, certain patterns may be considered more important than others. For example, frequent patterns in a query workload, for tuning the summary. Our preference value for a pattern captures this notion.

Definition 9. Preference Value of a Pattern. Given a set of patterns $P = (p_1, p_2, \ldots, p_m)$ with frequencies (freq(p_1), freq(p_2), …, freq(p_m)). Let $P_{PI} = (p_{PI1}, p_{PI2}, \ldots, p_{PIm})$ be a vector such that $0 \leq p_{PIi} \leq 1$ for every $p_{PIi} \in p_{PI}$. If p_{PIi} defines the importance of pattern p_i, we define the preference value of p_i (p_{PVi}) as the number of patterns which are less important than p_i, i.e., number of patterns p_j in P such that $p_{PIi} > p_{PIj}$.

In Definition 9 above, we do not assume any particular technique for computing the importance of a pattern. However for the purpose of tuning the summary to favour frequent patterns, the importance of a pattern can simply be computed as the ratio of its frequency to that of the most frequent patternyes. Our testing did not include preference values, so we will defer further treatment of them until the appendix.

To motivate the estimation value of patterns, we note that if there is a match for a pattern p where sequence(p) = $e_1, e_2, \ldots, e_k$ in the tree, its frequency freq(p), is the integer associated with the matched node labeled e_k. If e_k is pruned in the tree, we guess freq(p) from freq(p′), where sequence (p′) = $e_1, e_2, \ldots, e_{k-1}$ is the parent of p in the tree, under the assumption that children of p′ in the tree have a uniform frequency distribution. Thus if p′ has m children with total frequency N, the frequency of each child is estimated as N/m. When the children of p′ are contracted, we compute and associate the ratio N/(m × freq(p′)) with p′. We keep this ratio (rounded to the nearest integer) and not N/m for ease of frequency propagation as we will discuss later. We refer to this ratio as the *growth rate* of p′, denoted by p'_{GR}. We compute one growth rate for p′ for all its children, to avoid overly increasing the size of the tree as patterns are pruned.

Our estimate for a contracted pattern p will be inaccurate if the frequency distribution of the children of its parent p′ is not uniform. To exploit the uniformity assumption, we attempt to prune the P-Tree by deleting the children of patterns for which the assumption holds. To do this, we let the probability of the occurrence of any child p of a pattern p′ be its proportion with respect to the total frequency of all children of p′ (i.e. freq(p)/N, where N is the sum of freq(p_j) for all p_j in children(p′)). If the random variable Y defines the occurrence of a child of p′, we can measure the evenness of the probability distribution of Y using its *entropy* [45] $H(Pr_Y)$ given by:

$$H(Pr_Y) = - \sum_j Pr_Y(p_j)\log_2(Pr_Y(p_j))$$

The entropy of a probability distribution gives a sense of the unpredictability of any value from the distribution. The entropy of a probability distribution is maximized if the distribution is uniform. Thus, to measure how uniformly distributed Y is, we normalize $H(Pr_Y)$ by dividing by its maximum entropy. We denote this ratio as p'_{ENT} for pattern P′ in the tree. If p′ has one child, we set p'_{ENT} to 1.

Definition 10. cEstimation Value of a Pattern. Given a set of patterns $P = (p_1, p_2, \ldots, p_m)$ with frequencies $(freq(p_1), freq(p_2), \ldots, freq(p_m))$ and some $\delta \geq 0$, the estimation value of p_i (p_{EVi}) is given by:

$$P_{ENT\,i} \; \frac{\left(\left| \{ p_j \mid p_j \in \text{children}(P_i) \text{ and } |(\text{freq}(p_i)\, p_{GRi}) - \text{freq}(p_j)| \leq \delta \} \right| \right)^h}{\left| \text{children}(P_i) \right|}$$

By definition, for a pattern p, p_{ENT} is at most 1. It is 1, if the frequency distribution of the children of p is uniform. If the exponent h is set to 1, the second term of the product measures how closely p estimates all its children with at most δ error. Thus if p_i and p_j both have three children, if p_i estimates only two within δ error while p_j estimates just one also within δ, this value will be higher for p_i (2/3) than for p_j (1/3). However, if p_i has six children and estimates only two within δ error, then the value will be 1/3 for both p_i and p_j, although p_i estimates more of its children outside δ than p_j. To cope with this, we set h to 1.5 so that the numerator will not overly dominate the denominator. To find the optimal value for δ i.e. the smallest integer for which the pruned P-Tree fits the given budget, we recursively increase a small integer exponentially in powers of 2 until enough patterns can be pruned to meet the budget. Suppose this occurs at $\varepsilon = 2^i$, then we find the optimal δ which lies between 2^{i-1} and 2^i using binary search. We now show how the observed and estimation values of patterns are combined to obtain a single value with which the P-Tree is pruned.

As we noted before, when the size of the unpruned P-Tree exceeds the budget, we reduce the size of the P-Tree by systematically selecting some nodes of the P-Tree which are to be pruned. We compute the value of each node of the P-Tree (the combination of its preference and estimation values, or simply the latter if the former is not given) and select nodes to be pruned as shown in the *Prune P-Tree* algorithm shown below.

```
1.  Prune P-Tree
2.  Input:      P-Tree T, Budget B, constant c
3.  Output: Pruned P-Tree T
4.  $\delta \leftarrow 0$; inc $\leftarrow 0$; Set estimable $\leftarrow \emptyset$; done $\leftarrow$ false
5.  while done = false do
6.      estimable $\leftarrow \emptyset$
7.      for each internal node v in T in bottom – up order do
8.          compute estimation value at $\delta$
9.          if v's estimation value > 0 then
10.             insert all its children into estimable
11.         end if
12.     end for
13.     if sizeof(T) - sizeof(estimable) $\leq$ B and $\delta$ is optimal then
14.         compute estimation values of patterns in T
15.         prune all patterns in estimable
16.         done $\leftarrow$ true
17.     else
18.         adjust $\delta$
19.     end if
20. end while
```

Figure 4.4: Pattern Tree Construction Algorithm

As lines 4-8 of the Prune-Tree algorithm show, estimation values are used to select the patterns to be pruned. The running time of the algorithm $O(Ld^{maxL}\log(\max_i\{freq(p_i)\}))$, is reasonable since logarithm is a slowly growing function and maxL will typically be small. The running time stems from the loops in lines 7-12 and 2-17. Lines 7-12 run in $O(Ld^{maxL})$ time, where L and d are the numbers of unique edge labels and the maximum degree of nodes in the graph respectively. Recall that the root of the P-Tree has a child for each unique edge label in the graph while internal nodes have at most d children. Lines 5-20 will be executed at most $\log(\max_i\{freq(P_i)\})$ times, since all patterns are estimable at $\delta = \max_i\{freq(P_i)\}$.

We illustrate the pruning process with an example. In the example below, we assume that information about the preference values of patterns are not provided.

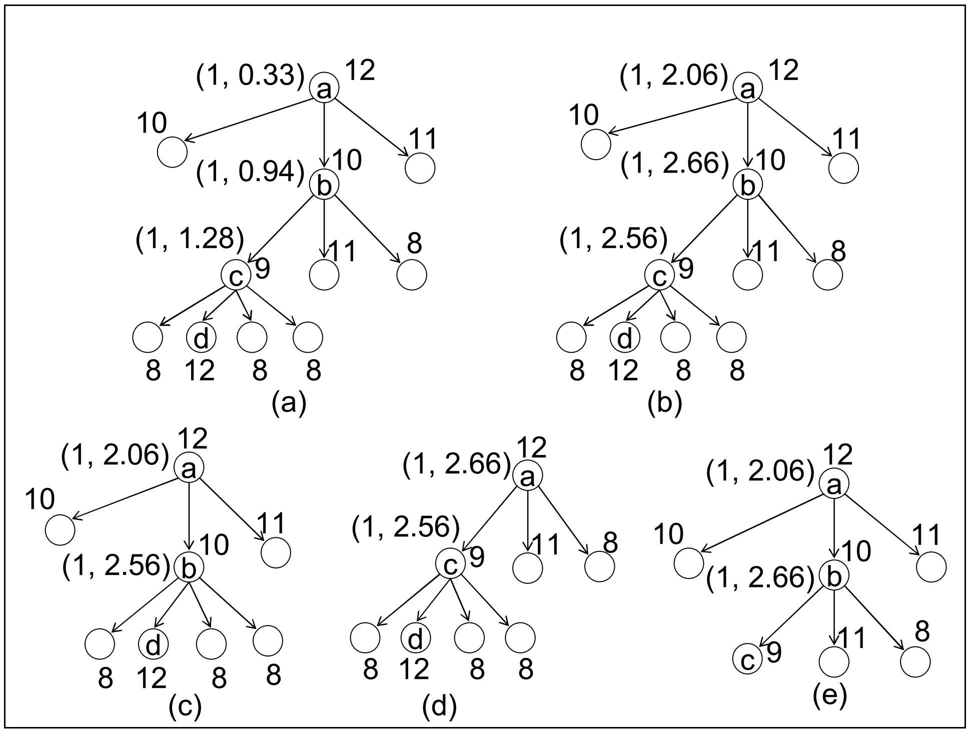


Figure 4.5: Pruning Nodes of the Pattern Tree

Example 4. *Figure 4.5a shows a subtree of a P-Tree. The 2-tuple (P_Z, P_V) associated with each internal node is its growth rate and its value, computed at $\delta = 1$ and c = 0 and assuming no importance information is given so that P_{PVi} is zero for all patterns. We show how the values are computed using node "a". Its growth rate is given by (10+10+11)/(3×12), which rounds to 1, so its frequency (12) estimates that of one child (11). With exponent 1.5, the second term of the equation in Definition 16 is 0.333. The entropy of the frequency distribution of its children is given by 10/31 × $\log_2(31/10)$ + 10/31 × $\log_2(31/10)$ + 11/31×$\log_2(31/11)$, or 1.583, with a maximum entropy ($\log_2(3)$) of 1.585 and ratio 0.999. Its estimation value is then 0.999(0.333) = 0.333. In figure 4.5b, the values are computed at $\delta = 2$. In figures 4.5c, d, and e, the children of* *nodes b, a, and c have been pruned at $\delta = 2$, with total estimation errors of 4, 5, and 6. Our technique will result in the pruning of figure 4.5c since b has the largest value.*

We note that when a set of internal nodes children(P_i) with parent P_i are to be pruned, if the children of any node in children(P_i) have been pruned, the average of the growth rates of such nodes are computed and associated with P_i. Thus, a node P in the pruned P-Tree may have at most maxL growth rates, ordered in increasing order of the original depths of their source in the P-Tree.

Frequency Estimation Using the Pattern Tree.

Having discussed the construction and pruning technique of the P-Tree, we now discuss how to estimate the frequencies of patterns of at most maxL from the P-Tree.

Given a pattern $G_P = (V_P, E_P, \lambda_P, P)$, we obtain its edge sequence $p = e_1, e_2, \ldots, e_k$ and check that each pair of edges e_i and e_j in p is connected in the semantic and structural summary. If they are, we match p against the P-Tree. If we find a complete match for p in the Pattern Tree, we return the frequency of the matched node e_k in the P-Tree. If we find a partial match, we consider the last matched node v_j in the P-Tree. If it matches e_k, we return its associated frequency which is the exact cardinality of p if no descendant of v_j was pruned. If it matches e_i $i < k$, we use its frequency to estimate that of the pruned node which originally matched e_k. We note that estimating the frequency of e_k requires estimating and propagating those of its k-i-1 immediate pruned ancestors. If $\xi_1, \xi_2, \ldots, \xi_r$, $k \le r \le maxL$ are the growth rates associated with v_j and freq(p_j) is the frequency associated with v_j, we estimate the frequency of e_k as:

$$P_{Fj} \times \prod_{r=1}^{k-i} \xi_r$$

For ease of exposition, we illustrate the estimation process with an example.

Example 5. *To estimate the frequency of pattern (a, b, c, d) using the pruned P-Tree of figure 4.5c, we find the partial match (a, b, _, d), we return 12 since node d is matched. With the P-Tree of figure 4.5e, we find the match (a, b, c, _). We return 9, since the growth rate of c is 1.*

4.3. Maximal Dependence Tree (MD-Tree)

To construct our MD-Tree summary, we adapted the Maximal Dependence Decomposition (MDD) technique that was proposed in [10]. The MDD was proposed as a technique that captures the most informative dependencies that exists amongst a set of DNA sequences of the same length, where it is impossible, due to limited data, to obtain a satisfactory estimate of all dependencies in the sequences. In our setting, we observe that even if it is possible to obtain accurate estimates of all dependencies that exist amongst a set of patterns of the same length, the space overhead that will be incurred in maintaining these dependencies will be very large. Our goal therefore, is to capture the most informative dependencies that exist amongst patterns of the same length, while incurring as small a space overhead as possible.

To motivate the MD-Tree, we observe that edges in certain positions in patterns may largely determine the probabilities of the occurrence of the patterns. For example, the three patterns of length 3 in figure 4.2d have the edge in the first (leftmost) position in common so that the edge in the second (middle) position, likely exerts a greater influence on the frequency of each of these patterns. The idea behind our adaptation of the MDD [10] approach then, is to assess the edge position with the greatest influence on the frequencies of the patterns. If no position of great influence exists, we assume that edges occur independently at each position. Thus, given a set of patterns of length at most maxL, we capture the probabilities with which edges occur at any position $p \leq maxL$ in the patterns in a tree data structure, which we call the MD-Tree. The next paragraphs discuss how to construct the MD-Tree through an adaptation of the Maximal Dependence Decomposition [10] technique. First, we give notations used in this discussion.

Notations.

In addition to the notations introduced in section 4.2, we now introduce more notations used in the discussions in this section. Given an instance graph $G_I = (V_I, E_I, \lambda_I, \tau, I, P, L)$, we let $\mathbf{N_E}$ denote the number of unique edge labels in G, i.e., N_E is size of the mapping $(\lambda_I(e), \tau(u), \tau(v))$, for each edge $e = (u, v)$ in E_I. Further, we let β be an integer in the range [1, 3] such that: 1) β is 1 if all edges in E_I are forward edges (in a depth first search traversal of the graph) only, 2) β is 2 if in addition to forward edges, E_I also contains backward edges only, 3) β is 3 if in addition to forward edges, E_I also contains self loop edges only or both backward and self loop edges. This is based on the assumption that a graph which contains a self loop edge is likely to contain a backward edge. We denote the probability with which edge e_i (its label and directionality) occurs at position j in the pattern by $\mathbf{Pr(e_i, j)}$.

Having established the notations used in the discussion, we begin the discussion of the MD-Tree by introducing our notion of *edge occurrence probability matrix*. Given a set P_K of patterns of length k, suppose it is known that edges occur independently at any position in the patterns, then we can estimate the frequency of any pattern P_j with edge sequence $(e_1, e_2, \ldots, e_k)$ as the product of the total frequency of patterns in P_K and the probability with which an edge e_i occurs at position j that is:

$$\mathrm{freq}(P_j) = \mathrm{freq}(P_K) \times \Pr(e_1, 1) \times \Pr(e_2, 2) \times \ldots \times \Pr(e_k, k)$$

To this effect, we pre-compute and maintain these probabilities in a probability matrix where $\Pr(e_i, j)$ is given by the ratio of the total frequencies of patterns in P_K such that edge e_i occurs at position j and the total frequencies of all patterns in P_K, i.e.,

$$\Pr(e_i, j) = \frac{\text{freq}(\{p \mid p \in P_K \text{ and edge } e_i \text{ occurs at position } j\})}{\text{freq}(P_K)}$$

Definition 11. Edge Occurrence Probability Matrix. A probability matrix PM_K for a set P_K of patterns of length k with $k > 0$, is a $\beta N_E \times k$ matrix whose rows represent the possible edge sub-patterns that may appear in any pattern in P_K and whose columns represent the positions in which the edge sub-patterns may occur. The $(i, j)^{th}$ entry of PM_K contains the probability that edge e_i occurs at position j.

To construct the probability matrix PM_K for patterns in P_K, we first obtain the row indices by assigning unique integer ids to the possible edge sub-patterns that may appear in any pattern of length at least two, in multiples of β, corresponding with the possible edge directions (forward, backward and self-loop). To ascertain that edge patterns are uniquely identified, we assign integer x to edge sub-pattern e_i such that if x modulo β is zero, one or two then x identifies e_i in the forward direction, backward direction or self-loop, respectively. The column indices are simply the k positions in which edge patterns may occur. Next, if edge sub-pattern e_i is assigned integer x, and e_i occurs at position j, we store $\Pr(e_i, j)$ in cell (x, j) of PM_K. Example 6 explains how the probability matrix in figure 4.6c is created

Example 6. *Figure 4.6a below shows the set P_2 of patterns of length two from figure 4.2d. The dimension of the probability matrix PM_2 for patterns in P_2 is 6×2 (i.e. β is 2, since there are no self loop edges). To construct PM_2, we assign integer ids to the edge types (5, 1, 4), (6, 3, 2) and*

(7, 4, 2) as shown in figure 4.6b. Next, we compute the entries for each cell (i, j) in PM_2 *as shown in figure 4.6c. Thus, cell (1, 1) holds the probability that edge type (5, 1, 4) occurs in a forward direction at position 1 in the patterns i.e. 6/7, cell (2, 2) holds the probability that edge type (5, 1, 4) occurs in a backward direction at position 2 of the patterns i.e. 3/7 etc. Under the independence assumption, the frequency of the pattern (1, 2, 5, 1, 4)(3, 2, 5, 1, 4) is estimated as 7(6/7)(2/7) i.e. 12/7, which rounds to 2.*

(1, 2, 5, 1, 4) (3, 2, 5, 1, 4)	3
(1, 2, 5, 1, 4) (2, 3, 7, 4, 2)	3
(1, 2, 6, 3, 2) (3, 2, 7, 4, 2)	1

(a)

1	(5, 1, 4)
2	(5, 1, 4)
3	(6, 3, 2)
4	(6, 3, 2)
5	(7, 4, 2)
6	(7, 4, 2)

(b)

	1	2
1	6/7	0
2	0	3/7
3	1/7	0
4	0	0
5	0	3/7
6	0	1/7

(c)

Figure 4.6: A Probability Matrix for Patterns of Length 2

If the independence assumption holds for all sets of patterns of length k, $1 \leq k \leq \text{maxL}$, then probability matrices PM_1, PM_2, ..., PM_{maxL} will suffice for estimating the frequency of these patterns. This assumption gives rise to our notion of the *Base MD-Tree*.

Definition 12. Base MD-Tree. Given the sets P_1, P_2, …, P_{maxL} of patterns of length at most maxL, a base MD-Tree for the patterns in P_i $1 \leq i \leq \text{maxL}$ is a triple (R_T, V_T, E_T) where $R_T \in V_T$ is the root of the tree and V_T and E_T are the sets of nodes and edges of the tree, such that $|V_T - R_T|$

= $|E_T|$ = maxL. All nodes in V_T - R_T are ordered children of R_T such that child i is associated with the probability matrix PM_i, for patterns in P_i. Each edge (R_T, i) is labeled with freq(P_i), the total frequency of all patterns in P_i.

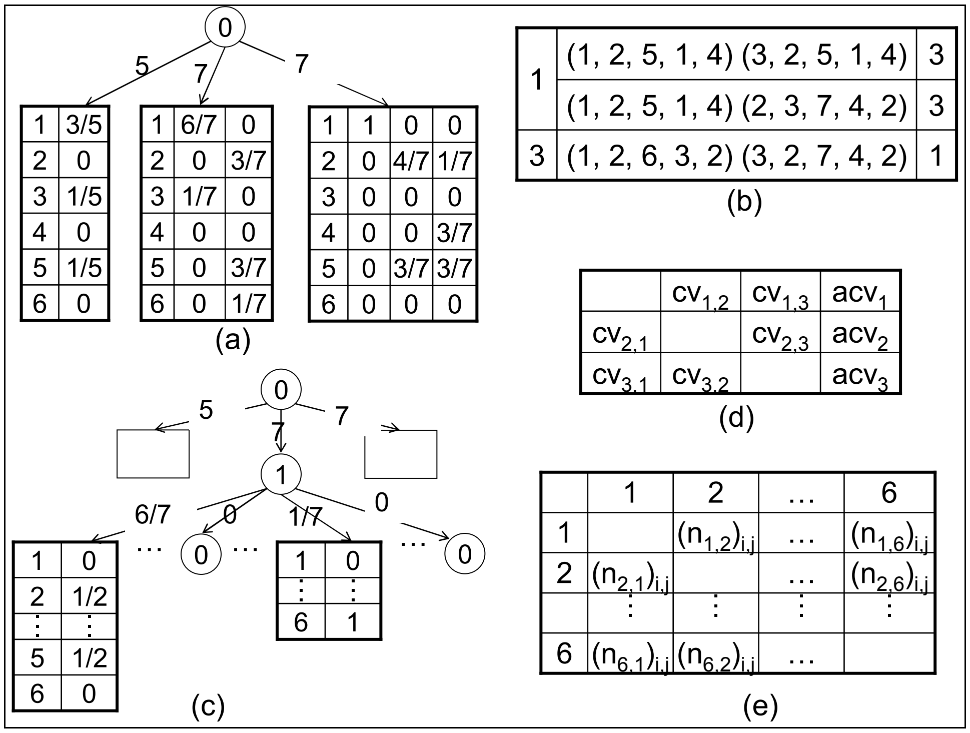
Example 7. *Figure 4.7a shows the base MD-Tree for the patterns of figure 4.2d.*

Under the assumption that edge types appear independently in patterns, the base MD-Tree holds sufficient information for estimating the frequency of patterns. As we saw in Example 6, this assumption may not always hold. Thus a refinement process on the base MD-Tree is required to capture dependency information that may exist among edge types with respect to the positions at which they occur in the patterns.

Given P_K, the set of all patterns of length k, suppose it is known that the occurrence of any edge at position i, $1 \le i \le k$ and $i \ne m$, depends on the edge at position m. We estimate the frequency of a pattern $P_j = (e_1, e_2, \ldots, e_k)$ in P_K as:

$$\mathrm{freq}(P_K) \times \Pr(e_m, m) \times \prod_{i=1, i \ne m}^{k} \Pr((e_i, i) \mid (e_m, m))$$

where $\Pr((e_i, i) \mid (e_m, m))$ is the conditional probability that e_i occurs at position i given that e_m occurred at position m. To refine the base MD-Tree to reflect this dependence, let node v_K be the node in the base MD-Tree that is associated with probability matrix PM_K. We create βN_E ordered children nodes rooted at v_K, where each child is associated with a new probability matrix PM_K of dimension $\beta NE \times$ k-1. Cell (a, b) of the probability matrix associated with child i of v_K contains the conditional probabilities that the edge type with integer id "a" occurs at position "b" given that the edge type with integer id "i" occurs at position m. The ith edge from v_K to its ith child is labeled with the probability that the edge type with integer id "i" occurs at position m in patterns in P_K. After the refinement process, the probability matrix associated with node v_K is deleted and the integer “m” is associated with v_K.



1	3/5	
2	0	
3	1/5	
4	0	
5	1/5	
6	0	

1	6/7	0
2	0	3/7
3	1/7	0
4	0	0
5	0	3/7
6	0	1/7

1	1	0	0
2	0	4/7	1/7
3	0	0	0
4	0	0	3/7
5	0	3/7	3/7
6	0	0	0

1	(1, 2, 5, 1, 4) (3, 2, 5, 1, 4)	3
	(1, 2, 5, 1, 4) (2, 3, 7, 4, 2)	3
3	(1, 2, 6, 3, 2) (3, 2, 7, 4, 2)	1

	$cv_{1,2}$	$cv_{1,3}$	acv_1
$cv_{2,1}$		$cv_{2,3}$	acv_2
$cv_{3,1}$	$cv_{3,2}$		acv_3

1	0	
2	1/2	
⋮	⋮	
5	1/2	
6	0	

1	0
⋮	⋮
6	1

	1	2	…	6
1		$(n_{1,2})_{i,j}$	…	$(n_{1,6})_{i,j}$
2	$(n_{2,1})_{i,j}$		…	$(n_{2,6})_{i,j}$
	⋮	⋮	⋮	⋮
6	$(n_{6,1})_{i,j}$	$(n_{6,2})_{i,j}$	…	

Figure 4.7: Refining the Base MD-Tree

For ease of exposition, we illustrate the refinement process with an example.

Example 8. *Given the base MD-Tree of figure 4.7a, suppose we know that edge types occurring at position 1 for patterns of length 2 has the greatest influence on the edge types at position 2, we refine the second child node (which we denote as v_2 in this example) of the root of the base MD-Tree as follows. First, we create 6 new children nodes for node v_2, one for each edge type. Next, we obtain the length 2 patterns which were used to create the probability matrix associated with v_2 i.e. the patterns shown in figure 4.6a. Next, we partition these patterns with respect to the occurrence of the 6 edge types at position 1. As shown in figure 4.7b, only the partitions for edge types 1 and 3 are non-empty. Using the patterns in the partition 1 and 3, we* *create two new probability matrices which we associate with child nodes 1 and 3 of v_2 respectively. Figure 4.7c shows the MD-Tree after the refinement process.*

When one or more nodes of a base MD-Tree are refined as described above, we call the modified tree a *refined MD-Tree*.

Definition 13. Refined MD-Tree. A refined MD-Tree for the sets $P_1, P_2, \ldots, P_{\text{maxL}}$ of patterns of length at most maxL, is a triple (R_T, V_T, E_T) where $R_T \in V_T$ is the root of the tree, V_T and E_T are the sets of nodes and edges of the tree respectively. The set of nodes V_T can be partitioned into two disjoint non-empty sets V_{Tleaf} and $V_{\text{Tnon-leaf}}$ such that every node in $v \in V_{\text{Tleaf}}$ or $V_{\text{Tnon-leaf}}$ is a leaf node or non-leaf node respectively. Further, a node v is associated with a probability matrix if and only if v is a leaf node and every non-leaf node v, $v \neq R_T$ has exactly βN_E children. The root node R_T has exactly maxL children.

Finding the Position of Maximal Dependence.

As we noted earlier, we refine a node of the base MD-Tree by determining the position in patterns of length k that exerts the greatest influence on the others. We now discuss how we determine this position.

Given the set P_K of patterns of length k, we find the position that greatly influences the others by performing chi-square association tests for edge types that occur at all pairs of positions i and j, $1 \leq i,j \leq k$, $i \neq j$, using the chi-square test statistic given by:

$$\sum_{m=1}^{\beta N_E}\sum_{n=1}^{\beta N_E}\frac{(O_{m,n} - E_{m,n})^2}{E_{m,n}}$$

In the equation above, $O_{m,n}$ is the sum of the frequency of patterns in P_K for which edge types m and n occur at positions i and j respectively and $E_{m,n}$, the expected mean of $O_{m,n}$ is given by:

$$\left(\sum_{a=1}^{\beta N_E} O_{m,a} \sum_{b=1}^{\beta N_E} O_{b,n}\right) \Big/ \sum_{a=1}^{\beta N_E} \sum_{b=1}^{\beta N_E} O_{a,b}$$

Thus, for the set P_K of patterns of length k, we find the position that greatly influences the others by performing chi-square association tests for edge types which occur at all pairs of positions i and j, $1 \le i,j \le k$, $i \ne j$. For each position i, we sum the values obtained from the chi-square association tests for positions i and j, $1 \le j \le k$, $j \ne i$. We refer to this sum as the *aggregated chi-square value* (*ACV*) of position i. If m is the position which has the maximum ACV (MACV), then position m exerts the greatest influence on all other positions if at least one of the chi-square values of positions m and j, $1 \le j \le k$, $j \ne m$, is statistically significant i.e. if the probability that positions m and j are associated by chance is less than a given level of significance. We clarify this with an illustration.

Example 9. *Suppose we wish to find a position of maximal dependence for a set of patterns of length 3. First, we create a 3×4 matrix as shown in figure 4.7d. To compute $CV_{1,2}$ for example, we create a 6×6 matrix as shown in figure 4.7e, such that cell i,j contains the number of times the edge types with integer ids i and j occur at positions 1 and 2 in the patterns respectively. $CV_{1,2}$ is then the value of the chi-square test statistic for the 6×6 matrix. Next, we store the sum of all entries of each row of the 3×4 matrix to the fourth column, then we find the maximum of the sums over the three rows. Suppose the maximum sum is ACV_2, we conclude that position 2 has the greatest influence on others but only if at least one of $CV_{2,j}$ is statistically significant*

Definition 14. Significant Node. Let v be a leaf node of a base or refined MD-Tree T, let PM_K be the probability matrix associated with v and let P_K be the set of patterns from which PM_K was created. We say that v is a significant node in T if there is a position j of maximal dependence in the patterns in P_K.

Definition 15. Complete MD-Tree. A complete MD-Tree is a refined MD-Tree T that has no significant leaf nodes.

Finding the Optimal MD-Tree.

From the foregoing discussion, it is obvious that the complete MD-Tree is the ideal choice amongst MD-Trees for obtaining accurate estimates of the frequency of patterns. However, the size of the complete MD-Tree may exceed the budget. Thus, given the sets of patterns P_1, P_2, .., P_{maxL}, our *optimal MD-Tree* is a refined MD-Tree whose size does not exceed B and from which the best estimates of the frequency of patterns in P_1, P_2, .., P_{maxL} can be obtained. The idea is to choose a sub-tree of the complete MD-Tree that fits the budget and maximizes the MACV values of the refined nodes in the sub-tree. Our objective of selecting nodes with high MACV values is because a high MACV value indicates a strong position of maximal dependence. Given a complete MD-Tree (r_T, V_T, E_T), let S = (s_{v1}, s_{v2}, …, s_{vm}) be the size increment induced on the MD-Tree when node v_i was refined. Note that s_{vi} is zero for the root and leaf nodes since they are not refined. Also let I = (i_{v1}, i_{v2}, … i_{vm}) be the impact of node v_i, given by $MACV_i/C_{vi}$, rounded to the nearest integer, where C_v is the number of columns of the probability matrix associated with v_i. We normalize the MACV values to avoid favoring nodes of larger patterns. Note also that i_{vj} is zero for the root and leaf nodes since they are not refined. The problem is to find a tree $T' = (V', E')$, $V' \subseteq V_T$ and $E' \subseteq E_T$ rooted at r_T, such that $\sum_j (s_{vj}) \leq B$ and $\sum_j (i_{vj})$ is maximized. This problem is an instance of the Tree Knapsack Problem (TKP) which is known to be NP-hard. Given x_j, an indicator variable with value 1 if v_j is selected as part of the optimal solution or 0 otherwise, TKP is formulated as the following integer programming problem: Maximize

$$\sum_{j}^{m} i_{vj} x_j$$

constrained on

$$\sum_{j}^{m} S_{vj} x_j \leq B,\ x_{pred(j)} \geq x_j$$

where pred(j) denotes the predecessor (parent) of j in T′. With this reformulation, several pseudo-polynomial time solutions based on dynamic programming (DP) have been proposed such as [13] with O(|V′|B) running time. We employ a greedy approximation with O(|V′|) running time. Given the complete MD-Tree, the vectors S and I and the summary size budget B, our greedy approximation creates the tree T′ = (V′, E′) by choosing maximal impact subtrees of T′ which fit the budget, as shown in the *Prune MD-Tree* algorithm.

```
1.  Prune MD-Tree
2.  Input: Tree Knapsack TK, Budget B
3.  Output: Pruned Tree TK
4.  V = (v1, v2, …, vm) //vector of nodes of TK in decreasing order of impacts
5.  tree_capacity ← 0
6.  while V not empty do
7.        v ← extract first element of V
8.        v_ancestors_size ← sizes of v and its ancestors
9.        if tree_capacity + v_ancestors_size ≤ B then
10.              initialize node v′ to v
11.              while u ≠ $r_T$ do
12.                    contract v′ into u for edge (u, v′)
13.                          delete v′ from V
14.               increment tree_capacity with v_ancestors_size
15.       else
16.              prune subtree at v, delete its nodes from V
17.end while
```

Figure 4.8: MD-Tree Construction Algorithm

Frequency Estimation Using the MD-Tree.

Given an optimal MD-Tree (r_T, V_T, E_T), we define λ_V as a function that maps nodes in V_T to integers (for internal nodes) or probability matrices (for leaf nodes) with which they are associated. The integers associated with internal nodes denote the position at which the node was split. We also define λ_E as a function which maps edges in E_T to integers (for edges emanating from the root) or real numbers (for all other edges) with which they are associated. The integer on edge i emanating from the root denotes the total frequency of patterns of length i, while the real numbers on all other edges denote the conditional probability of the occurrence of its incident node given the edge pattern at the split position. We also define the function *id* on edge patterns that return the integer id assigned to the edge type of the pattern.

Let P = e_1, e_2, …, e_k be the edge sequence of a graph pattern $G_P = (V_P, E_P, \lambda_P, P)$ of length k. To estimate the frequency of P, we check that each pair of edges e_i and e_j in P is connected in the semantic and structural summary. If so, beginning from the kth child (say v) of r_T, we estimate freq(P) as:

$$\mathrm{freq}(P) = \lambda_E(r_T, v_1)\left(\prod_{i=1}^{j} \lambda_E(v_i, v_{i+1})_{id(e_{\lambda_V(v_i)})}\right)\left(\prod_{r=1,\, r\notin S}^{k} (\lambda_v(v_{j+1}))_{(id(e_r),\, r)}\right)$$

In the product above, given an edge $(v, v')_r$ subscripted with r, the subscript r denotes the rth edge of node v. The integer j is the number of edges of the optimal MD-Tree found on the path from the root to a leaf node as defined by the subscripts on the edges, so that the node v_{j+1} is a leaf. The subcripts (r, r′) are integer indices for accessing cell (r, r′) of the probability matrix associated with node $\lambda_V(v_{j+1})$. The set *S* holds labels of all nodes on the path from R_T to v_{j+1} so that at v_{j+1}, any integer in the range [1, maxL] not in the set *S* did not label any node on this path. We keep elements of *S* sorted as they are inserted, so that at the leaf v_{j+1}, we can check for elements of [1, maxL] not contained in *S* in O(maxL) time. The depth of the MD-Tree is at most maxL; thus, the time complexity for estimating pattern frequencies is given by:

$$O(\text{maxL} \log(\text{maxL}))$$

We illustrate the estimation process by an example.

Example 10. *To estimate the frequency of the pattern $P = e_1, e_2$ of length 2 given by (1,2,5,1,4) (3,2,5,1,4) from the MD-Tree of figure 4.7, we first access the second child node of the root. Let this node be denoted v_1. Since $\lambda_V(v_1) = 1$, we insert 1 into set S and set the frequency of P (freq(P)) to freq(P_2) which is 7. Recall from figure 4.6b that id(e_1) = id(1, 2, 5, 1, 4) is 1. So, we access the node on which the first edge of v_1 is incident. Let this node be v_2. Next, we multiply freq(P) by $\lambda_E(v_1, v_2)$ given by 6/7, resulting in 6. Then, we obtain $\lambda_V(v_2)$ which yields the probability matrix PM_2 associated with v_2 which must have only one column. Since the set S contains the integer 1, the lone column of PM_2 must index position 2 of patterns in P_2. Further, id(e_2) = id(3,2,5,1,4) is 2, thus we access cell which represents the index (2, 2) in PM_2 to obtain 1/2. We then multiply freq(P) which is currently 6 by 1/2 to obtain 3, the final estimate for the frequency of P.*

4.4. Estimating the Frequency of Large Patterns

Given a large pattern whose length is greater than maxL, the idea is to use smaller patterns whose frequencies are known, to estimate the frequency of the large pattern. Thus we break up the large pattern into smaller patterns whose frequencies are known and then combine the frequencies of the small patterns to obtain an estimate for that of the large pattern. There are several possible ways of breaking up the large pattern starting from one in which the smaller patterns are disjoint to one in which the smaller patterns have overlaps in all but one edge. In [1], the authors demonstrated that breaking up a large path into non-disjoint smaller paths that intersect in all but one edge work well in estimating the frequency of large path. Further, the authors in [17] have also shown that breaking up a large twig into non-disjoint smaller twigs which intersect in all but one edge work well in estimating the frequency of the large twig. In the light of these works, we estimate the frequency of a large pattern by breaking it up into non-disjoint smaller patterns which intersect in all but one edge. We break up a large pattern by a traversal that visits all edges in the pattern as few times as possible. More formally, given a large pattern $G_P = (V_P, E_P, \lambda_P, P)$ with $|E_P| > maxL$, as always, we check that its edge sequence $p = e_1, e_2, \ldots, e_k$ is connected in the semantic and structural summary. If so, we break up G_P into $G'_1, G'_2, \ldots, G'_{|E|-maxL+1}$ non-disjoint connected patterns such that G'_i intersects G'_{i-1} in all but one edge. Let G''_i denote the intersecting edges of G'_i and G'_{i-1}. Next, we obtain the edge sequences $p'_1, p'_2, \ldots, p'_{|E|-maxL+1}$ and $p''_2, p''_3, \ldots, p''_{|E|-maxL+1}$ for the patterns $G'_1, G'_2, \ldots, G'_{|E|-maxL+1}$ and $G''_2, G''_3, \ldots, G''_{|E|-maxL+1}$, respectively. Like in [17], we assume conditional independence to estimate the frequency of G_P as follows:

$$\mathrm{freq}(p) = \mathrm{freq}(p'_1) \times \prod_{r=2}^{|E|+\mathrm{maxL}-1} \frac{\mathrm{freq}(p'_r)}{\mathrm{freq}(p''_r)}$$

Since G_P may be broken up into $G'_1, G'_2, \ldots, G'_{|E|-maxL+1}$ in several different ways, we select that for which frequency estimates of the patterns $p'_1, p'_2, \ldots, p'_{|E|-maxL+1}$ are obtained along the deepest paths, i.e., paths which have the maximum total split nodes in the MD-Tree or along paths with the least pruned nodes in the P-Tree.

Example 11. *Figure 4.9a shows a large pattern of length 4. The exact matchings of this pattern in the sample graph database shown in figure 4.9b is 4. Suppose maxL is 2, we estimate this pattern by subdividing it into the sub-patterns (a) 1—head→2,2—teacherOf→3, (b) 2—teacherOf→3,2—author→4, (c) 2—author→4,4—inJournal→5. Sub-pattern (a) has 3* *matchings, (b) has 5 matchings and (c) also has 5 matchings. Sub-pattern (a) overlaps with (b) on the edge 2—teacherOf→3 which has 3 matchings, while sub-pattern (b) overlaps with (c) on the edge 2—author→4 which has 6 matchings. Therefore, the estimate for the pattern is 3(5/3)(5/6) which is 4.167 which rounds to 4.*

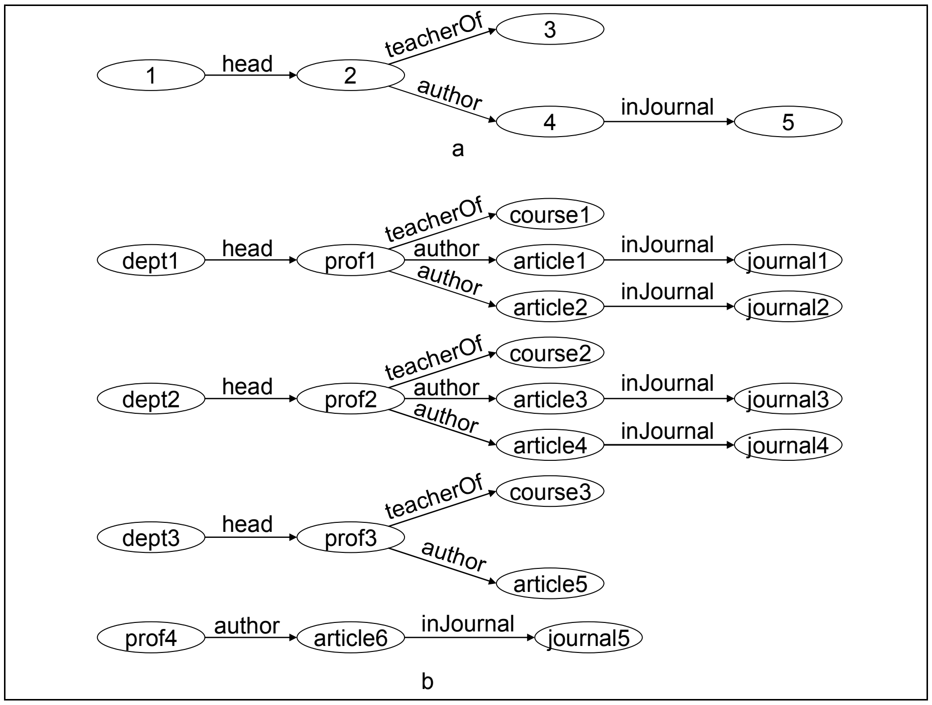


Figure 4.9: (a) A Large pattern and (b) A Sample Graph Database

Using the example shown in figure 4.9, we compare our combining function to the traditional database join formula given by $N_A N_B/\max(V_A, V_B)$ where N_A is the number of tuples in relation A, N_B is the number of tuples in relation B, V_A and V_B are the numbers of distinct join attribute values in relation A and relation B, respectively.

Example 12. *For the traditional database formula, there are several ways to subdivide the pattern of figure 4.9a. We show only 2 ways here. One possible subdivision is 1—head→2,2—teacherOf→3 and 2—author→4,4—inJournal→5. The first sub-pattern has 3 matchings, while* *the second sub-pattern has 5 matchings and node 2 which is the join node (prof) has 3 distinct matchings for both. Applying the formula, we have 3(5/3) which is 5. Another possible subdivision is 1—head→2,2—teacherOf→3, 2—author→4 and 4—inJournal→5. The first sub-pattern has 3 matchings, the second (author relation) has 6 matchings and the third (inJournal relation) has 5 matchings. Node 2 (prof) which is the join node for the first and second sub-patterns has 3 matchings for the first and 4 matchings for the second. Node 4 (article) which is the join node for the second and third sub-patterns has 6 matchings for the second and 5 matchings for the third. Applying the formula recursively we have 3(6/4)(5/6) which is 3.75 which rounds to 4.*

5. Experimental Evaluation

In this section, we present the results of the experimental study we conducted based on the proposed techniques described in this thesis. The goal of our experimental evaluation is to show that the estimates obtained from the proposed summaries are indeed more accurate than commonly used techniques and more importantly effective for optimizing join queries posed over RDF graphs. We also demonstrate that the pruned summaries perform nearly as well as the unpruned summaries in terms of accuracy and query optimization. We show this by comparing the running time of several join queries optimized using our techniques to those optimized using other techniques.

5.1. Methodology

Techniques. We considered the following techniques in our experimental study, depending on how cardinality estimates are obtained for query optimization:

- Proposed Graph Summaries (GSummaries): We implemented the P-Tree and MD-Tree summaries proposed in this work. We implemented a main memory execution engine that uses the classic dynamic programming technique [44] for enumerating query plans. We used the schema-aware model [51] for shredding RDF graphs into relational tables.
- Simple Greedy Technique (SGT): We implemented a simple greedy technique that builds a left deep query plan by beginning with the relationship with the smallest cardinality and extending to its adjacent relationships using an algorithm similar to that for growing a minimum spanning tree.

- Traditional Query Optimization Technique (TQOT): We also implemented another execution engine that simulates a main memory relational database execution, using the same dynamic programming enumeration as the execution engine for our graph summaries. The difference however is in the way the result cardinality estimates are obtained. While for our proposed techniques, the graph summaries supply the cardinalities, for the main memory relational database, we used the well known uniformity assumptions employed by relational databases to compute the cardinality estimates. If n_1 and n_2 are the sizes of the two relations to be joined and if v_1 and v_2 are the number of distinct values of the join attributes, the cardinality of the join of the two relations is estimated as $n_1n_2/\max\{v_1, v_2\}$

Implementation Details. We implemented the proposed techniques (P-Tree and MD-Tree summaries) in C++ with experiments performed on a 1.8GHz Dual AMD Opteron processors and 16GB RAM. We created sparse matrices using sparseLib++ [25] libraries and used BRAHMS [23] to parse the graphs. We used the schema-aware model for shredding RDF graphs into tables and the B+Tree implementation of [22] for indexes. To compare the techniques, we implemented the joins using hash join algorithm with intermediate tables materialized in memory. We also implemented the joins using piped iterators on left deep plans only. Although our work is mainly targeted towards optimizing join queries without constraints, in our experiments, we show that with a combination of equi-depth [26] histograms and our proposed summaries, our techniques perform well for join queries with uri constraints or literal constraints. The histograms provide cardinality information for the constraints, based on which we compute the selectivity of the constraint, which we propagate to estimates of unconstrained joins from our graph summaries. If the value obtained from the histogram for a particular constrained relationship is f and the cardinality of that relationship when unconstrained is n, we compute the selectivity as *f/n*. If the estimate of an unconstrained join from our graph summaries is *c*, we then propagate the constraint to the unconstrained join by the product *c(f/n*).

Datasets. For our experiments, we used one synthetic dataset and one real life dataset. The real life dataset which we used is the Mondial dataset. The Mondial dataset is a rich compilation of geographical Web data sources on global statistics of world countries, cities, provinces, seas, and international organizations. We converted an OWL version of the Mondial dataset into RDF using protégé. For the synthetic dataset, we further enriched the Lehigh University Benchmark (LUBM) dataset with new classes and relationships. To do so, we followed the model used by LUBM in determining the sizes of instances of classes and relationships such that the generated dataset models a real life university domain. For scalability experiments, we generated three LUBM datasets with 10, 15 and 20 universities. The table below shows the properties of these datasets.

Table 5.1: Dataset Properties

	Mondial	LUBM10	LUBM15	LUBM20
# Instance nodes	5841	248880	385530	528334
# Instance edges	18565	3021748	4677928	6416925
# Literal Nodes	12450	106300	164795	225209
# Literal edges	14002	429927	668110	915398
# Unique Instance edges	28	31	31	31

Summaries. Stored as a list, the size of all patterns of length up to three for the Mondial dataset is 378534 bytes and the size of the unpruned P-Tree and MD-Tree are 164792 and 128690 bytes, with the P-Tree giving about a 56% reduction in size and the MD-Tree giving about a 66%

reduction in size. We pruned the P-Tree and MD-Tree, by constructing summaries which are 50% of the original unpruned summary sizes.

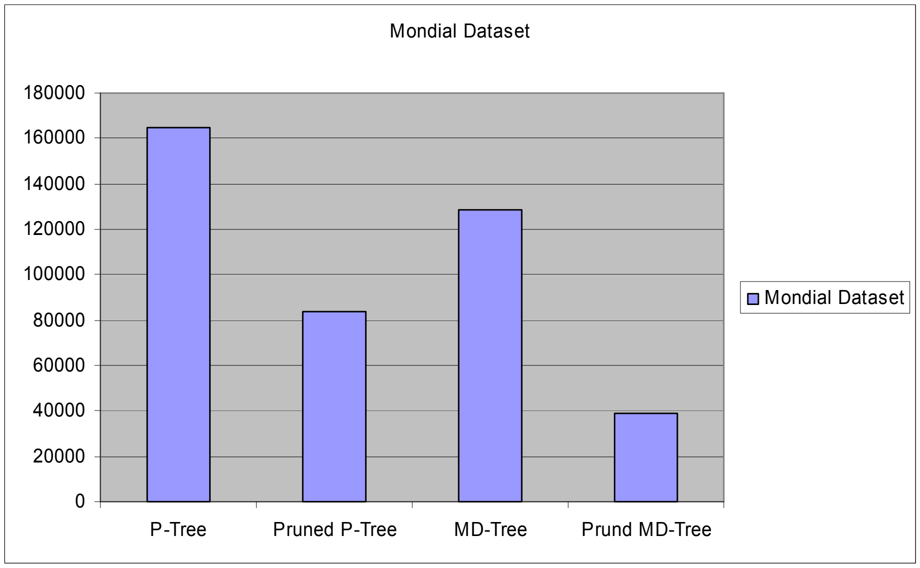


Figure 5.1: The sizes of the unpruned and pruned summaries for the Mondial dataset

On the other hand, the size (as a list) of all patterns of length two for the LUBM dataset is 9614 bytes. Its unpruned P-Tree and MD-Tree are 5582 and 9422 bytes, with the P-Tree giving about a 42% reduction in size and the MD-Tree giving a minimal reduction in size. Once again, we pruned the P-Tree and MD-Tree, by constructing summaries which are 50% of the original unpruned summaries. For both datasets, we used a 5% significance level and a β value of 3 for constructing the MD-Tree summaries.

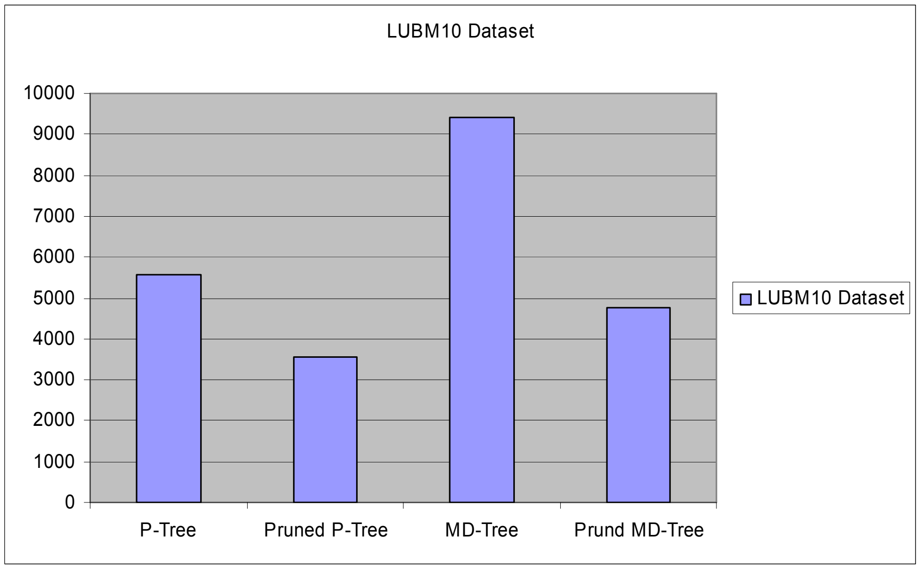


Figure 5.2: The sizes of the unpruned and pruned summaries for the LUBM10 dataset

Time Analysis. The time for discovering all patterns of length maxL is the most time-consuming part of our approach. Fortunately, it is a preprocessing step and depends on the connectedness of the dataset. The time needed for constructing the P-Tree and MD-Tree summaries is in the order of tenths of seconds.

Queries. As we noted earlier, our proposed summaries are targeted towards join queries posed over RDF databases where relationships are specified in the joins but without constraints on the nodes. However, in our experiments, we also evaluated the efficiency of our techniques when combined with histograms for join queries which may have constraints on the nodes. To make our evaluations as realistic as possible, we avoided choosing queries which are a random concatenation of relationships in the data graph. While doing so could have led us to test queries with larger sizes, we made the choice of queries which are meaningful and are as close as possible to real life queries that users are likely to pose over the datasets. First, we chose queries which have no constraints on the nodes, then we also chose queries which have uri constraints on the nodes as well as those which have literal constraints on the nodes. For this reason, our queries can be grouped into three categories. 1) Join queries with no constraints at all, i.e., for which relationships are specified and all nodes are variables. 2) Join queries with uri constraints, i.e., those for which relationships are specified but some nodes are constrained to be of a particular uri. 3) Join queries with literal constraints, i.e., those for which relationships are specified but some nodes are constrained to be a particular literal value. We have included the queries in SPARQL in the appendix of this paper. Queries 1 – 4 are posed over the Mondial dataset while queries 5 – 8 are posed over the LUBM dataset.

Error metric. We used the relative error metric *|freq(p) – freq(p^)|/freq(p)* to measure the estimation error where *freq(p)* and *freq(p^)* are the true and estimated frequencies of *p*. Since the nature of the SGT is such that it does not provide estimates for patterns, we do not include it in our comparison of the estimation error of the techniques.

Results. We first show the results of queries over the Mondial dataset for the hash join implementation with intermediate table materialization. Figure 5.3 shows the accuracy of the techniques for query 1, which is an unconstrained join query with three relationships. The cardinality of the result of this query is 4916. The unpruned P-Tree and the pruned P-Tree both estimate the cardinality of the result of this query as 4915 with an estimation error of 0.0002. The pruned P-Tree exhibits a good performance for this query. On the other hand, the unpruned MD-Tree estimates it as 4934 with an estimation error of 0.0037 while the pruned MD-Tree estimates it as 3276 with an estimation error of 0.3336. The TQOT estimates it as 3445 with an estimation error of 0.2992. The pruned MD-Tree estimates the cardinality of this query with the highest error amongst all the techniques. This is for the case where a maximal impact subtree is pruned because it does not fit the budget, leading to estimation using the independence assumption.

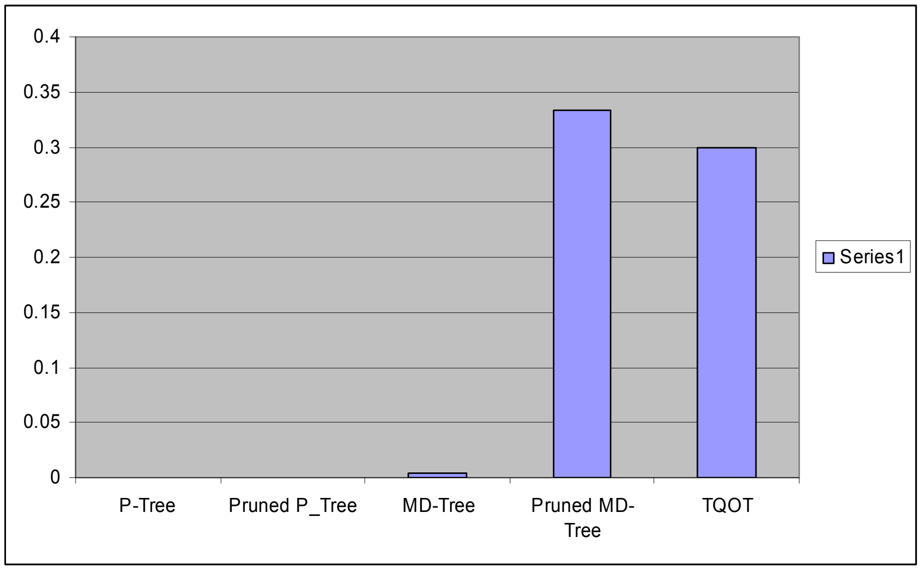


Figure 5.3: The estimation errors of the techniques for Query 1

As figure 5.4 shows, the unpruned P-Tree, pruned P-Tree, the unpruned MD-Tree and the TQOT all performed equally well while the pruned MD-Tree and SGT did not perform as well. We note that although the size of the pruned P-Tree is about half the size of the unpruned P-Tree, the pruned P-Tree has the same performance as the unpruned P-Tree. This is an indication of the efficiency of the pruning technique. As indicated by its estimation error, the performance of the pruned MD-Tree is not as good as its unpruned counterpart. As mentioned earlier, this is because of the pruning of a maximal impact subtree that does not fit the budget. For the TQOT, this is the case where the estimation error is not high enough to cause it to choose a sub-optimal plan.

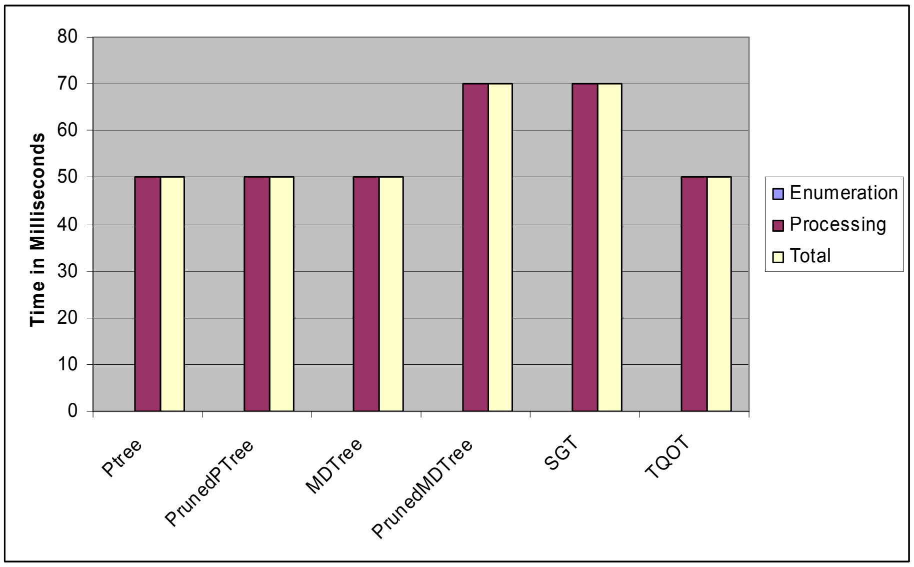


Figure 5.4: Result of Query 1: An Unconstrained Join Query over the Mondial Dataset

Figure 5.5 shows the estimation errors of the evaluated techniques for query 2, which is another unconstrained join query involving three relationships. For this query, the cardinality of the result is 877. The unpruned P-Tree and the pruned P-Tree both estimate the cardinality of the result as 865, with an estimation error of 0.0137. The effectiveness of the pruning technique of the P-Tree is shown by the fact that although the pruned P-Tree is about half the size of the unpruned P-Tree, its estimation error is the same as that of the unpruned P-Tree. The unpruned MD-Tree estimates it as 868 with an estimation error of 0.0103. The pruned MD-Tree estimates the result cardinality as 3276 with an estimation error of 2.7355, while the TQOT estimates the result cardinality as 2005 with an estimation error of 1.2862. Again, the unpruned P-Tree and pruned P-Tree have the best performance with the least estimation error, followed by the unpruned MD-Tree. The pruned MD-Tree has the worst performance

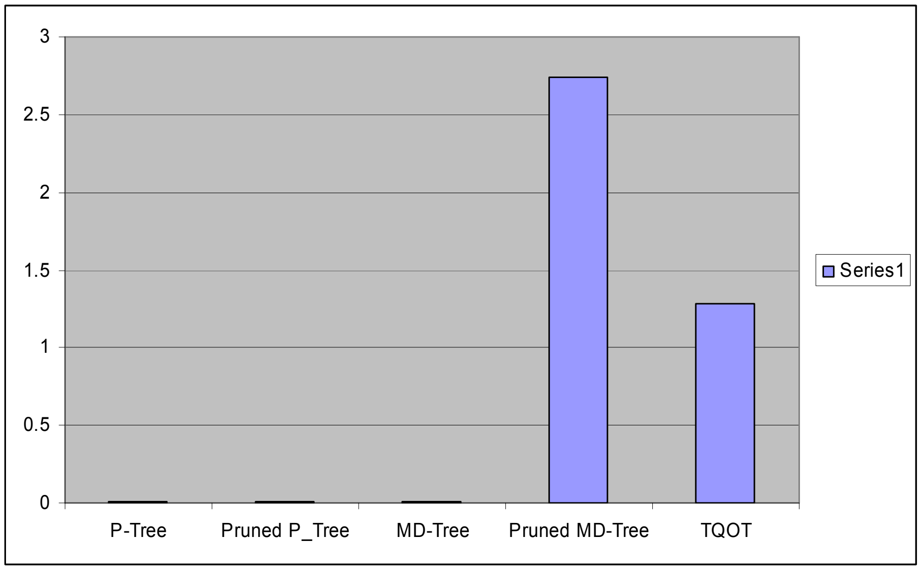


Figure 5.5: The estimation errors of the techniques for Query 2

As figure 5.6 shows, the TQOT exhibits the worst performance, while all the other techniques exhibit the same performance. For the TQOT, this is the case where the join uniformity assumption does not hold and the estimation errors are large enough to cause it to choose a sub-optimal query plan. Although the estimation error of the pruned MD-Tree is larger than that of the unpruned MD-Tree as well as the TQOT, the pruned MD-Tree still performs as well as the unpruned MD-Tree and better than the TQOT. This is because the intermediate estimates obtained from the pruned MD-Tree are more accurate than the overall estimate, as such it still finds the optimal plan.

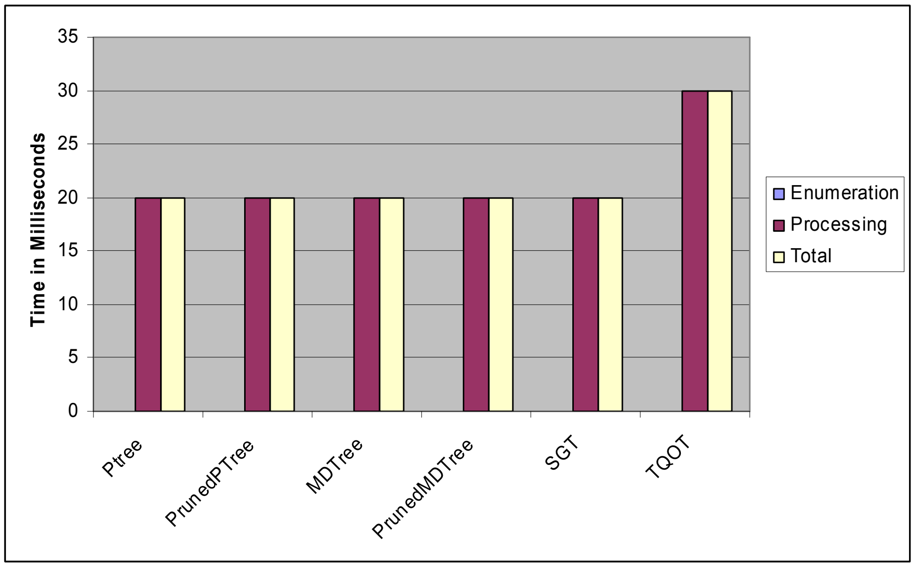


Figure 5.6: Result of Query 2: An Unconstrained Join Query over the Mondial Dataset

Figure 5.7 shows the estimation errors of the techniques for query 3, a join query involving three relationships, with a uri constraint. The cardinality of the result of this query is 1033. The unpruned P-Tree, pruned P-Tree and unpruned MD-Tree all estimate the cardinality of the result of this query as 429, with an estimation error of 0.5847. For this query, once again, the pruning technique for the P-Tree prunes effective as the estimation error for the P-Tree is the same as that of the unpruned P-Tree. The pruned MD-Tree estimates the cardinality of this query as 0 with an estimation error of 1. The estimation error of the pruned MD-Tree is greater than that of the unpruned MD-Tree, reflecting the case when a maximal impact subtree of the MD-Tree is pruned in order to meet the budget. The TQOT estimates it as 289, with an estimation error of 0.7202.

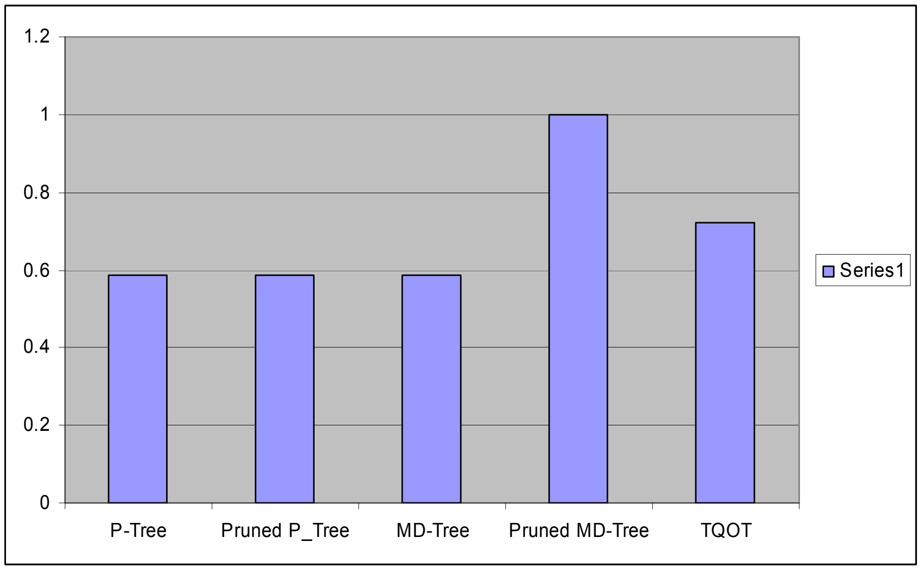


Figure 5.7: The estimation errors of the techniques for Query 3

Figure 5.8 shows the query processing performances of the techniques for query 3. For this query, all the techniques find the optimal query plan and thus exhibit the same performance. This is due to the constraint on the query which causes the techniques to find an optimal query plan, even though their estimation errors differ.

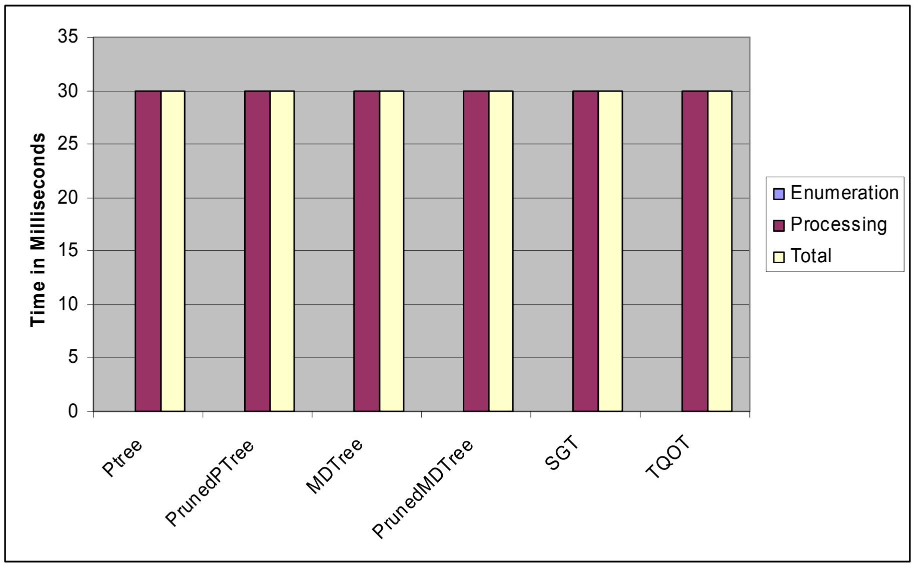


Figure 5.8: Result of Query 3: A Uri Constrained Join Query over the Mondial Dataset

Figure 5.9 shows the estimation errors of the evaluated techniques for query 4, a join query involving 4 relationships with a literal constraint. The cardinality of the result of this query is 162. The unpruned P-Tree, pruned P-Tree, unpruned MD-Tree and pruned MD-Tree all estimate the cardinality of the result as 1 with an estimation error of 0.9938. The TQOT estimates it as 1296 with an estimation error of 7. Recall that for queries with constraint, we propagate the selectivity of the constraint to the estimates of the joins. Our propagation assumes that a uniform number of patterns will be affected by the constraint. For this query, this uniform assumption does not hold and hence the larger errors in the estimates for the cardinality of the results.

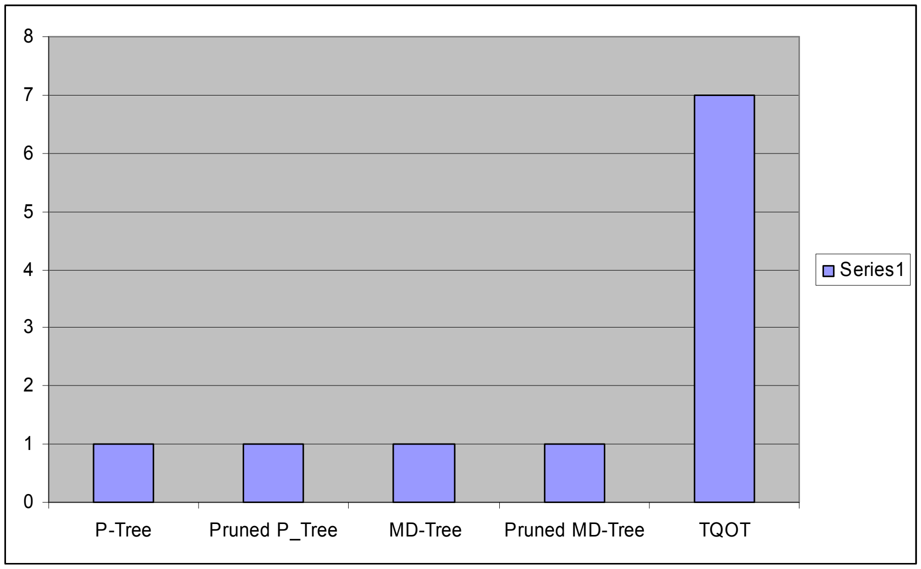


Figure 5.9: The estimation errors of the techniques for query 4

As figure 5.10 shows, for this query, both the SGT and TQOT techniques perform worse than our techniques. In conformity with the estimation errors of our techniques, all of our techniques exhibit the same performance in terms of the time spent for query processing. For this query, the effectiveness of our pruning techniques is once again shown by the fact that both the pruned P-Tree and pruned MD-Tree perform as well as their unpruned counterparts.

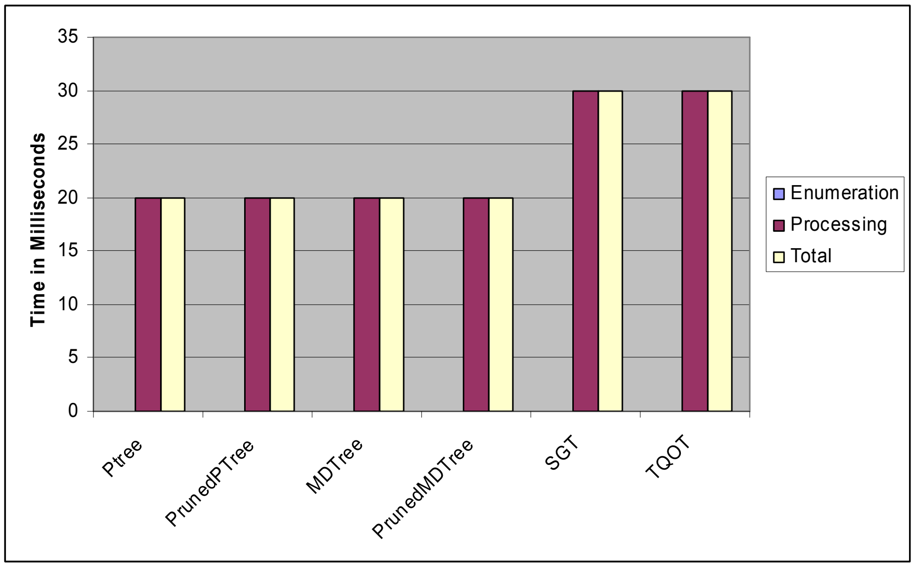


Figure 5.10: Result of Query 4: A Literal Constrained Join Query over the Mondial Dataset

In summary for the Mondial dataset, our experimental results show that for the accuracy of estimates of unconstrained join queries, our techniques perform better than the Traditional Query Optimization Technique with the unpruned P-Tree, pruned P-Tree and the unpruned MD-Tree having an average error of 0.7% compared to 79% for the Traditional Query Optimization Technique. The pruned MD-Tree performs worst with an average error of 153%. For the constrained queries, all our techniques perform much better than the Traditional Query Optimization Technique with the unpruned P-Tree, pruned P-Tree and the unpruned MD-Tree having an average error of 79%, while the pruned MD-Tree has an average error of 100% compared to 386% average error of the Traditional Query Optimization Technique. Over both the unconstrained and constrained join queries, our techniques are superior with an average error of 40% for the unpruned P-Tree, pruned P-Tree and the unpruned MD-Tree and 127% average error of the pruned MD-Tree compared to 233% average error of the Traditional Query Optimization Technique.

As can be observed from the foregoing discussion, the accuracy of the pruned P-Tree was the same as that of the unpruned P-Tree for the queries 1 – 4. To assess the level of degradation of the accuracy of the pruned P-Tree with respect to the size of the pruned P-Tree, we pruned the P-Tree to fit a 10KB budget, 25KB budget and 50KB budget. Figure 5.11 shows the estimation errors of the pruned P-Trees of sizes 10KB, 25KB and 50KB for query 2. As the figure shows, the pruned P-Tree of size 10KB shows a large estimation error for this query, while the pruned P-Tree of Size 25KB and 50KB have the same minimal estimation error (0.0137) as the unpruned P-Tree (see figure 5.5 above). As the figure shows, one can construct a summary that is as small as 25KB, without increasing the estimation error when compared to the unpruned P-Tree, for this query. As the figure also shows, it takes the same amount of time to process the query for each of the pruned summaries.

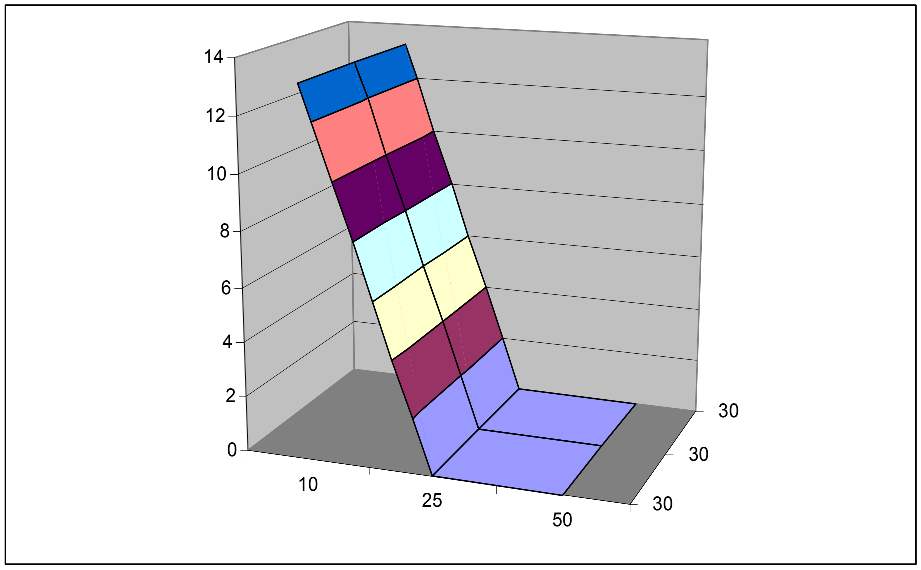


Figure 5.11: The Estimation Error for Query 2 for Summaries of Size 10KB, 25KB and 50KB

Figure 5.11 shows the estimation errors of the pruned P-Trees of sizes 10KB, 25KB and 50KB for query 3. Once again, the pruned P-Tree of size 10KB shows a large estimation error for this query, while the pruned P-Tree of Size 25KB and 50KB have the same minimal estimation error

(0.5847) as the unpruned P-Tree (see figure 5.7 above). Once again, as illustrated by this figure, for this query, we can construct a summary that is as small as 25KB, without increasing the estimation error when compared to the unpruned P-Tree. As figure 5.10 and figure 5.11 show, the estimation error increases as the size of the summary decreases. However, the estimation error of the summary does not overly increase for a reasonably small budget, in these cases a budget of size 25KB.

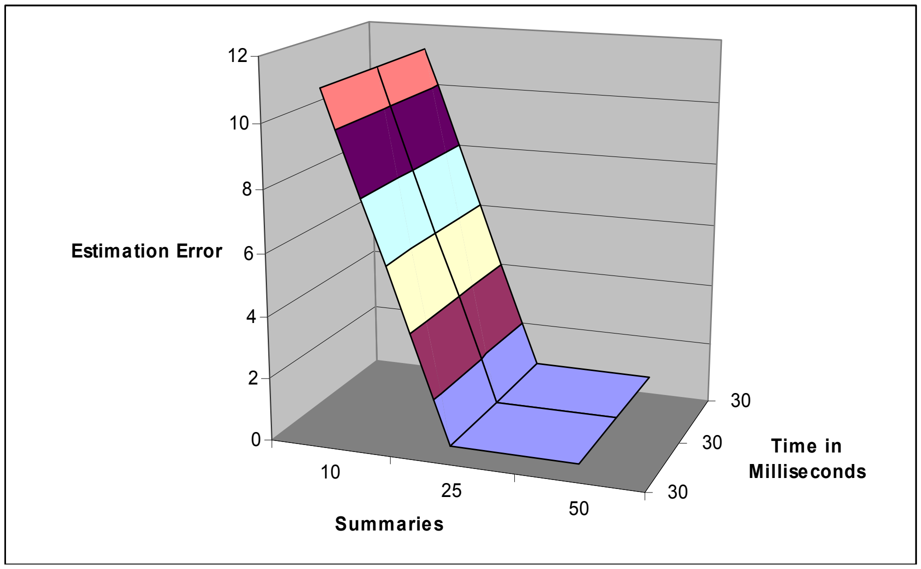


Figure 5.12: The Estimation Error for Query 3 for Summaries of Size 10KB, 25KB and 50KB

We now show the results over the LUBM dataset for the hash join implementation with intermediate table materialization. For this dataset, once again we posed queries in categories 1 – 3. Figure 5.13 shows the estimation errors of the evaluated techniques for query 5, an unconstrained join query involving three relationships, over the LUBM10 dataset. The cardinality of the result of this query is 2912. The unpruned P-Tree, the pruned P-Tree and the unpruned MD-Tree all estimate it as 2906 with an estimation error of 0.0021 while the pruned MD-Tree estimates it as 1 with an estimation error of 0.9997. This is because maximal impact subtrees along the path of this query were aggressively pruned causing estimation to be based on the independence assumption which does not hold in this case. The TQOT estimates it as 2914 with an estimation error of 0.0007.

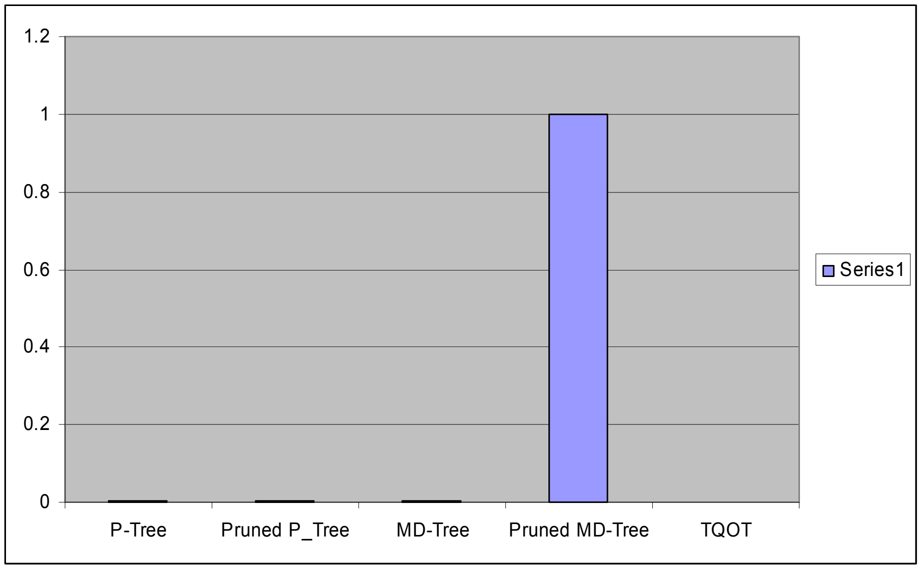


Figure 5.13: The estimation errors of the techniques for query 5

As figure 5.14 shows, our pruned P-Tree and pruned MD-Tree show encouraging performances with both performing as well as their unpruned counterparts. Once again, the performance of the pruned P-Tree and pruned MD-Tree show the effectiveness of the pruning techniques. Our techniques have the same performance as the TQOT, while the SGT exhibits a worse performance than all the other techniques. Further, the time taken to enumerate the plans in all cases is very negligible. To show how the techniques scale with larger dataset, we posed the same query over LUBM15 and LUBM20 datasets and as figure 5.15 shows, the techniques scale linearly with increase in dataset size and the performance of SGT gets even worse with increase in dataset size.

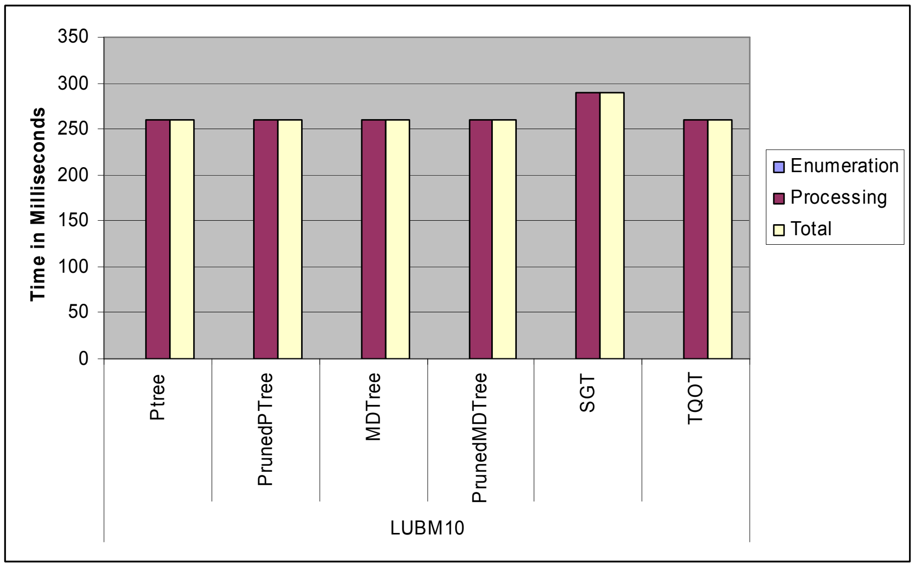


Figure 5.14: Result of Query 5: An Unconstrained Join Query over the LUBM10 Dataset

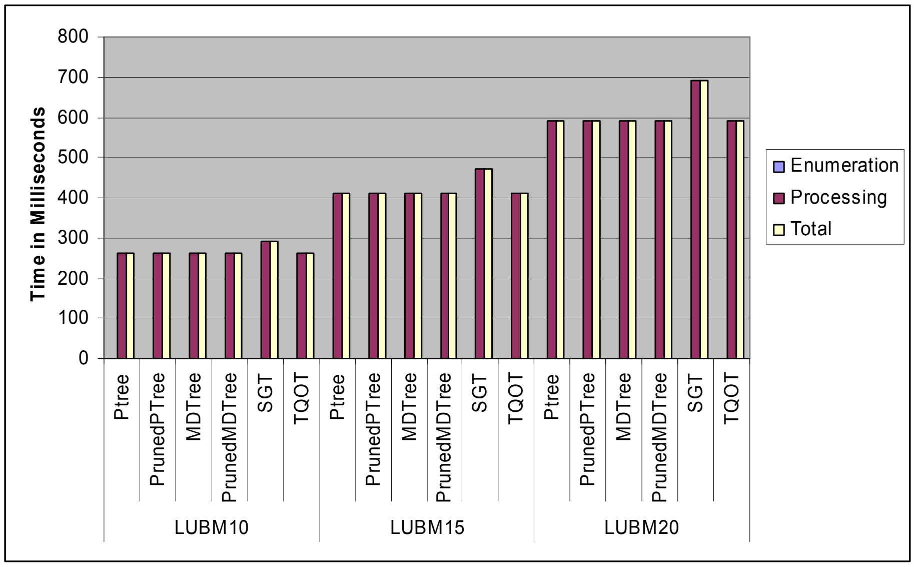


Figure 5.15: Result of Query 5: An Unconstrained Join Query over the LUBM10, LUBM15 and LUBM20 Datasets

Figure 5.16 shows the estimation errors of the techniques for query 6, another unconstrained join query involving four relationships, over the LUBM10 dataset. The cardinality of the result of this query is 559. The unpruned P-Tree, the pruned P-Tree and the unpruned MD-Tree all perfectly estimate the cardinality of the result as 559 with no estimation error. However, the pruned MD-Tree estimates it as 1 with an estimation error of 0.998. Once again, this is due to the aggressive pruning of maximal impact subtrees along the path of this query with estimation using the independence assumption which does not hold. The TQOT estimates the cardinality of the result as 3790 with an estimation error of 5.78.

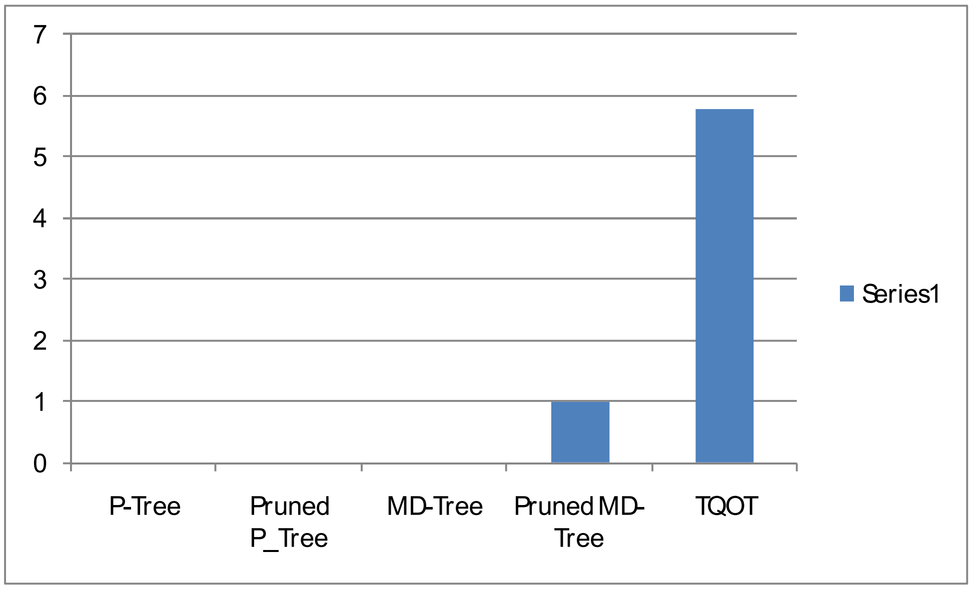


Figure 5.16: The estimation errors of the techniques for query 6

For this query, as shown in figure 5.17, in conformity with their estimation errors, our pruned P-Tree performed as well as its unpruned counterpart while the pruned MD-Tree shows a worse performance than its unpruned counterpart. Our techniques all perform better than the TQOT. This is due to the fact that the uniformity assumptions on which it is based do not hold and the estimation errors are large enough to cause it to choose a worse plan than the P-Tree and MD-Tree techniques. The SGT also exhibits the same performance as our techniques.

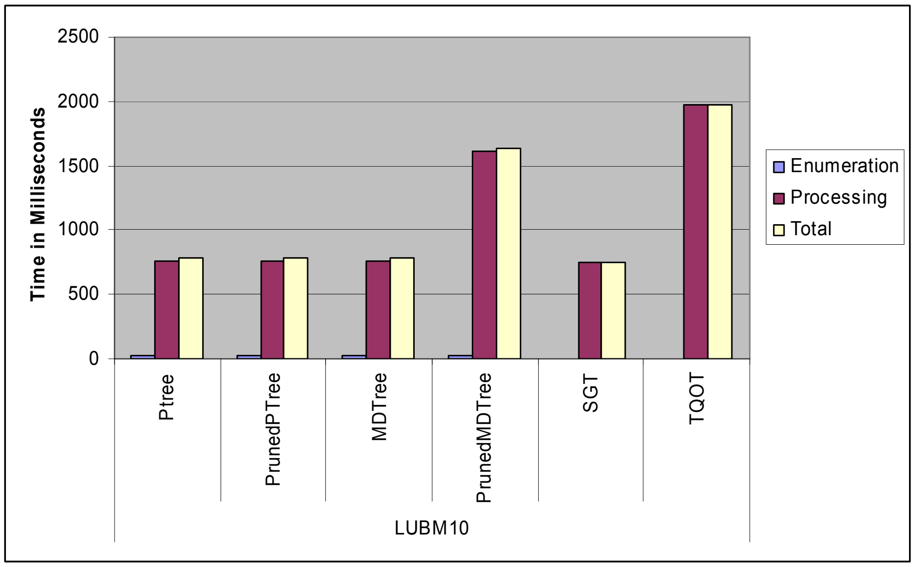


Figure 5.17: Result of Query 6: An Unconstrained Join Query over LUBM10 dataset

Figure 5.18 shows the estimation errors of the techniques for query 7, a uri constrained join query involving three relationships over the LUBM10 dataset. The cardinality of the result of this query is 9. The unpruned P-Tree estimates it as 1 with an estimation error of 0.8889, the pruned P-Tree estimates it as 7 with an estimation error of 0.2222. For this query, the estimate obtained from the pruned P-Tree is more accurate than that obtained from the unpruned P-Tree. This is as a result of the combination function that is used to obtain the estimate of a pattern which is larger than the size of patterns maintained in the P-Tree. The unpruned MD-Tree estimates it as 2 with an estimation error of 0.7778, while the pruned MD-Tree estimates it as 1 with an estimation error of 0.8889. The TQOT estimates it as 4 with an estimation error of 0.5556.

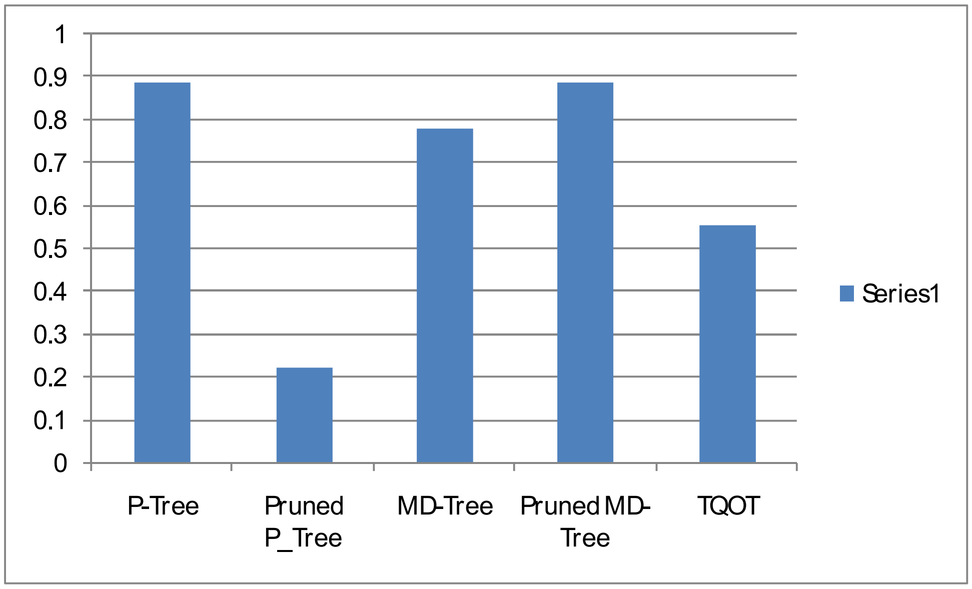


Figure 5.18: The estimation errors of the techniques for query 7

As shown in figure 5.19, all techniques except the SGT came up with the same query plan and thus have the same performance. This is because their absolute estimation errors are small and thus they all find an optimal plan.

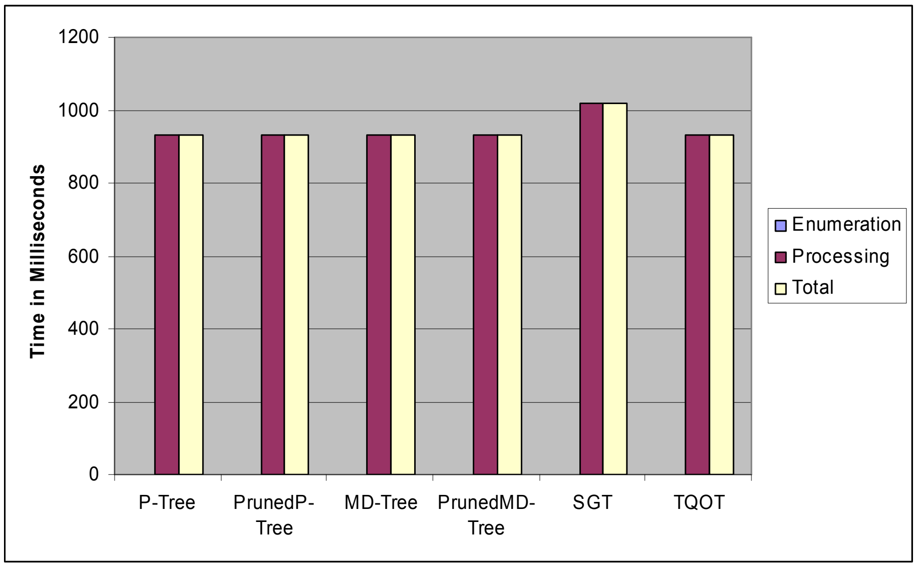


Figure 5.19: Result of Query 7: A Uri Constrained Join Query over LUBM10 Dataset

Figure 5.20 shows the estimation errors of the techniques for query 8, a literal constrained join query involving four relationships over the LUBM10 dataset. The cardinality of the result of this query is 93. The unpruned P-Tree, the pruned P-Tree, the unpruned MD-Tree and the pruned MD-Tree all estimate the cardinality of the result as 1 with an estimation error of 0.9892. Once again, this is because the uniform assumption based upon which the selectivity of the constraint is propagated to the estimates of the joins does not hold. The TQOT estimates the cardinality of the result as 341 with an estimation error of 2.6667.

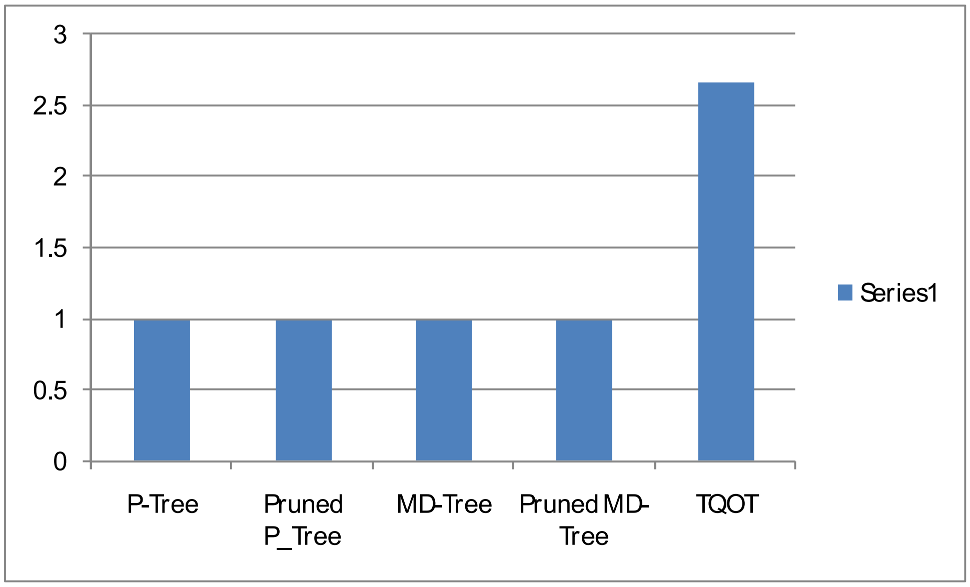


Figure 5.20: The estimation errors of the techniques for query 8

Although our techniques have a smaller overall estimation error for this query than the TQOT, the TQOT performs slightly better than our techniques, as figure 5.21 shows. This is because the estimates obtained for intermediate patterns from our techniques with the selectivity propagation, caused it to choose a worse plan than the TQOT. The SGT shows a slightly worse performance than all our techniques.

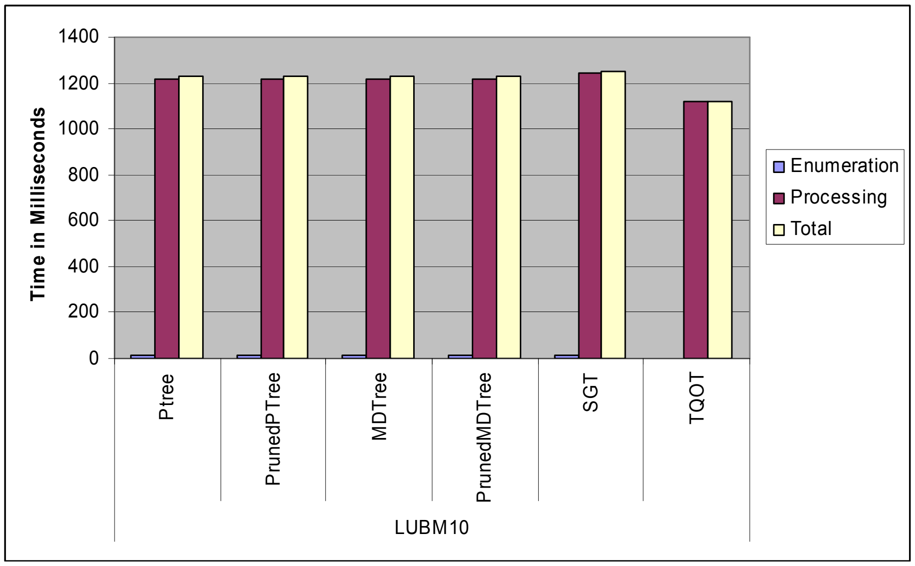


Figure 5.21: Result of Query 8: A Literal Constrained Join Query over LUBM10 Dataset

In summary for the LUBM dataset, our experimental results show that our techniques perform better on the average in terms of the accuracy of the estimates. For the unconstrained join queries, the unpruned P-Tree, pruned P-Tree and unpruned MD-Tree all had an average error of 0.11%, while the pruned MD-Tree had an average error of 100% compared to 289% average error of the Traditional Query Optimization Technique. For the constrained join queries, the unpruned P-Tree had an average estimation error of 94%, the pruned P-Tree had an average error of 61%, the unpruned MD-Tree had an average error of 88%, while the pruned MD-Tree had an average error of 94% compared to 161% average error of the Traditional Query Optimization Technique. Over both the unconstrained and constrained join queries, the unpruned P-Tree had an average error of 47%, the pruned P-Tree had an average error of 30%, the unpruned MD-Tree had an average error of 44%, while the pruned MD-Tree had an average error of 97% compared to 225% average error of the Traditional Query Optimization Technique

We now show the comparisons of the hash join implementation with intermediate table materialization versus the piped iterators on left deep plans only using indexed joins. We show this comparison only for the LUBM10 dataset.

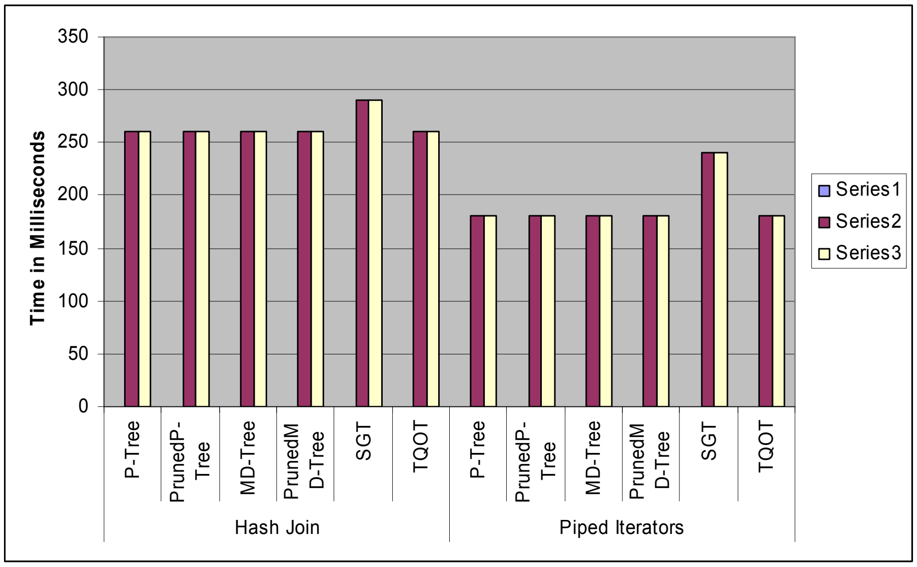


Figure 5.22: Comparison of the Hash Join Implementation with the Piped Iterator Implementation for Query 5

Figure 5.22 shows the result of query 5 (an unconstrained join query involving three relationships) for the join with intermediate table materialization and the piped iterator implementation. The techniques all show a reduced running time for the piped iterator implementation when compared with intermediate table materialization. In addition, one of the piped techniques, the Simple Greedy Technique (SGT), shows an increased running time when compared to the other piped techniques (P-Tree, pruned P-Tree, MD-Tree, pruned MD-Tree, TQOT). For this query, all the techniques except for the SGT find the same left deep plan and as such they all have the same running time.

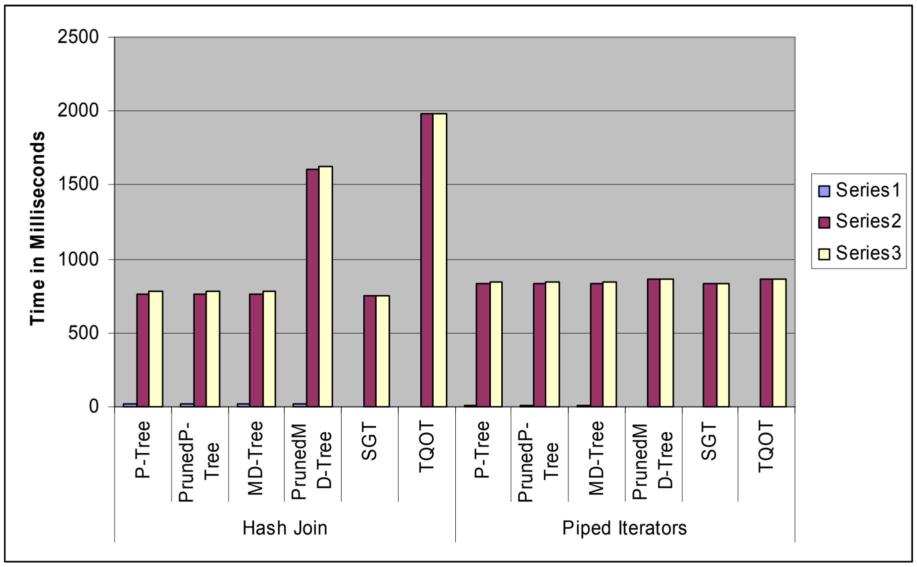


Figure 5.23: Comparison of the Hash Join Implementation with the Piped Iterator Implementation for Query 6

Figure 5.23 shows the results of the intermediate table materialization and the piped iterator implementation for query 6, an unconstrained join query involving four relationships. When compared with their corresponding intermediate table materialization implementation, the pruned MD-Tree and the TQOT show a reduced running time for the piped iterator implementation. However, the rest of the techniques show a very slight increase in the running time for the piped iterator implementation.

Figure 5.24 shows the result of the intermediate table materialization and the piped iterator implementation for query 7, a URI constrained join query involving three relationships. All the techniques show a reduced running time for the piped iterator implementation when compared with the intermediate table materialization. All the techniques, except the SGT, show extremely fast running times while the SGT shows an increase in the running time when compared to the other techniques for the piped iterator than for the intermediate table materialization.

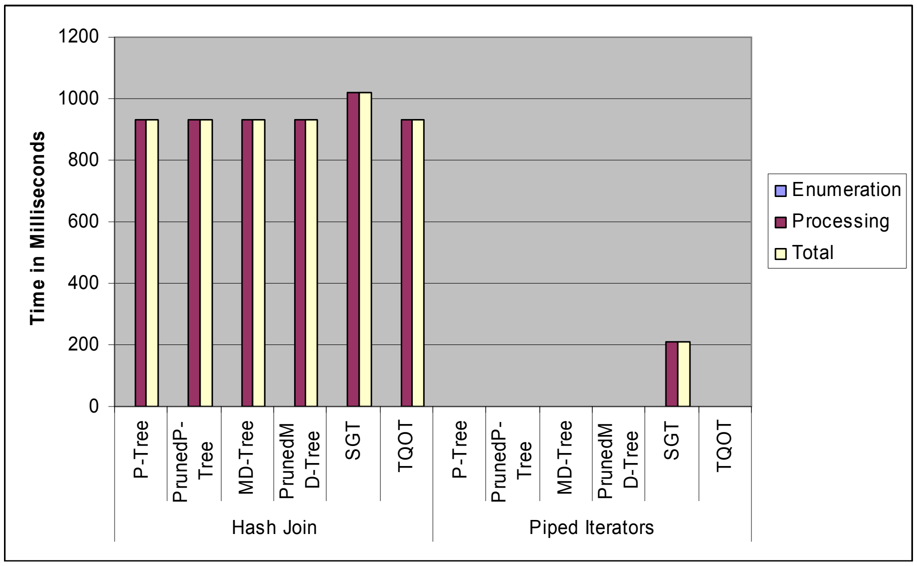


Figure 5.24: Comparison of the Hash Join Implementation with the Piped Iterator Implementation for Query 7

Figure 5.25 shows the result of query 8, a literal constrained join query involving four relationships, for the intermediate table materialization strategy and the piped iterator execution strategy. Compared to the intermediate table materialization strategy, all the techniques show a reduced running time in the piped iterator execution strategy. However, the SGT shows an increased running time when compared to the other techniques for the piped iterator execution strategy than for the intermediate table materialization strategy.

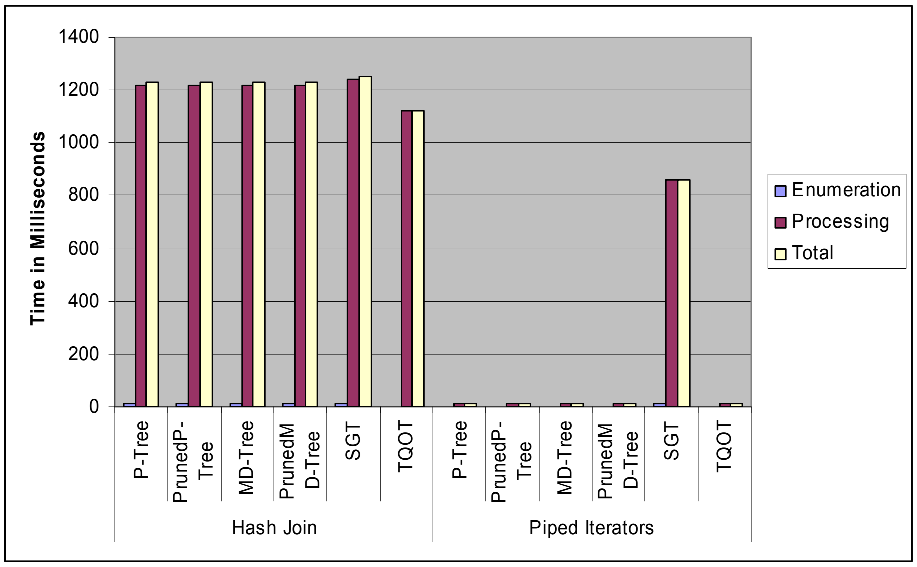


Figure 5.25: Comparison of the Hash Join Implementation with the Piped Iterator Implementation for Query 8

6. Conclusion and Future Work

Graph pattern querying is important for eliciting information from graphs. Optimizing graph pattern queries requires estimating the frequency of subgraphs in a query graph pattern. In this work, we presented two techniques for summarizing the structure of graphs and we showed how to prune the summary to fit a given space budget. As our experiments showed, for unconstrained join queries, the proposed P-Tree and MD-Tree exhibited encouraging performance when compared to the traditional query optimization techniques used in relational database systems and the simple greedy technique used by the execution engine of the SPARQLeR system. The pruned P-Tree is relatively stable for all datasets but performs best when graph patterns that share a common sub-graph pattern co-occur. The pruned MD-Tree performs best when single points of dependence exist among subgraphs. Our experiments also show that the time taken to enumerate plans is often negligible when compared to the actual query processing time. Although the Dynamic Programming approach arguably may take some time for enumeration as the number of relationships grows larger, the authors in [33] have shown a new class of enumeration algorithms that work to reduce the enumeration time without a large decrease in performance when compared to the dynamic programming enumeration.

There are several directions we hope to explore in the future. First, we will look into a summarization and estimation framework for RDF graphs which may have hierarchies on the edges, by virtue of the subpropertyOf property. Next we will extend our statistical graph summaries to better cope with constraints so as to avoid the inaccurate estimates obtained when propagating the selectivity of a constraint to the estimates of the patterns obtained from the summaries. Further we will investigate techniques for gracefully accommodating updates to the data graph into our summaries without a complete reconstruction of the summaries.

References

1. Aboulnaga, A., Alameldeen, A., Naughton, J.: Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In VLDB, 2001.

2. Aleman-Meza, B., Hakimpour, F., Arpinar, B. I., Sheth, A.: SwetoDblp Ontology of Computer Science Publications. Tech. Report, CS Department, University of Georgia, 2006

3. Anyanwu, K., Maduko, A., Sheth, A.: SemRank: Ranking Complex Relationship Search Results on the Semantic Web. In Proc. 14th WWW Conference, 2005.

4. Beckett, D.: RDF/XML Syntax Specification (Revised). W3C Recommendation. 10th February 2004. http://www.w3.org/TR/rdf-syntax-grammar/

5. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American, 2001.

6. Boulicaut, J., Bykowski, A, Rigotti, C.: Approximation of Frequency Queries By Means of Free-sets. In Proc. 4th European PKDD Conference, 2000.

7. Brickley, D., Guha, R. V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation. 10th February, 2004. http://www.w3.org/TR/rdf-schema

8. Broder, A. Z., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, T., Wiener, J.: Graph structure in the web. WWW9/Computer Networks, 2000.

9. Broekstra, J., Kampman, A., Harmelen, F. V.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Proc. 1st ISWC Conference, 2002

10. Burge, C. Identification of Complete Gene Structures in Human Genomic DNA. Ph.D. Thesis, Stanford University, Stanford, CA. 1997

11. Carroll, J. et al.: Implementing the Semantic Web Recommendations. In Proc. 13th WWW Conference, 2004.

12. Chen, Z., Jagadish, H. V., Korn, F., Koudas, N., Muthukrishnan, S., Ng, Raymond., Srivastava, D.: Counting Twig Matches in a Tree. Proc. 17th ICDE Conference 2001.

13. Cho, G., Shaw, D. X. A Depth-First Dynamic Programming Algorithm for the Tree Knapsack Problem. INFORMS J. Computing 9(4) 1997, 431-438.

14. Dehaspe, L., Toivonen, H., King, R. D.: Finding Frequent Substructures in Chemical Compounds. In KDD, 1998.

15. Deshpande, A., Garofalakis, M., Rastogi,R.: Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In Proc. 2001 ACM SIGMOD Conference, 2001.

16. Desphande, M. Kuramochi, M. Wale, N.: Frequent Substructure-Based Approaches for Classifying Chemical Compounds. In TKDE. Vol. 17, No. 8. Aug. 05.

17. Freire, J., Haritsa, J., Ramanath, M., Roy, P., Simeon, J.: StatiX: Making XML Count. In Proc. 2002 ACM SIGMOD Conference, 2002.

18. Getoor, L., Taskar, B., Koller, D.: Selectivity Estimation using Probabilistic Models. In Proc. 2001 ACM SIGMOD Conference, 2001.

19. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semi-Structured Databases. In Proc. 23rd VLDB Conference, 1997.

20. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2), 2005, pp158-182.

21. Harth, A., Decker. S.: Optimized index structures for querying RDF from the web. In LAWEB 2005

22. http://idlebox.net/2007/stx-btree/

23. http://lsdis.cs.uga.edu/projects/semdis/brahms

24. http://lsdis.cs.uga.edu/projects/semdis/swetodblp

25. http://math.nist.gov/sparselib++/

26. Ioannidis, y.: The History of Histograms (abridged). In Proc. 29th VLDB Conference, 2003.

27. Jagadish, H., Koudas, N., Muthukrishnan, S.: Optimal Histograms with Quality Guarantees. In Proc. 24th VLDB Conference, 1998

28. Janik, M., Kochut, K.: BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In Proc. 4th ISWC Conference, 2005.

29. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Schol, M.: RQL: A Declarative Query Language for RDF. In WWW'02, Honolulu, Hawaii, USA, May7-11 2002

30. Karvounarakis, G., Christophides, V., Plexousakis, D., Alexaki, S.: Querying RDF descriptions for community web portals. In Proc. French BDA Conference, 2001.

31. Klyne, G., Carroll, J. J.: RDF Concepts and Abstract Syntax. W3C Recommendation. (Revised) February 2004. http://www.w3.org/TR/rdf-syntax-grammar/

32. Kochut, K., Janik, M.: SPARQLeR: Extended SPARQL for Semantic Association Discovery. In ESWC 2007.

33. Kossmann, D., Stocker, K.: Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. In ACM Transactions on Databases. Volume 25, Issue 1 (March 2000)

34. Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Parr, R.: XPathLearner: An On-Line Self-Tuning Markov Histogram for XML Path Selectivity Estimation. In Proc. 28th VLDB Conference, 2002.

35. Magkanaraki, A., Karvounarakis, G., Christophides, V., Plexousakis, D., Anh, T.: Ontology Storage and Querying. Technical Report No 308, April 2002

36. McHugh, J., Widom, J.: Lore: Query Optimization for XML. In Proc. 25th VLDB Conference, 1999

37. Pei, J., Dong, G., Zou, W., Han, J.: On Computing Condensed Frequent Pattern Bases. In ICDM, 2002.

38. Perry, M. TOntoGen: A Synthetic Data Set Generator for Semantic Web Applications. In SIGSEMIS Bulletin.

39. Polyzotis, N., Garofalakis, M., Ioannidis, Y.: Selectivity Estimation for XML Twigs. In ICDE, 2004.

40. Polyzotis, N., Garofalakis, M.: Statistical Synopses for Graph-Structured XML Databases. In SIGMOD, 2002.

41. Poosala, V., Ioannidis, E.: Selectivity Estimation Without the Attribute Value Independence Assumption. In Proc. 23rd VLDB Conference, 1997.

42. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Working Draft. 19th April 2005. http://www.w3.org/TR/rdf-sparql-query/

43. Seaborne, A.: RDQL - A Query Language for RDF, WWW Consortium, Member Submission SUBM-RDQL-20040109, January 2004

44. Selinger, P. G., Astrahan, M. M., Lorie, R. A., Price, T. G.: Access Path Selection in a Relational Database Management System. In Proceedings of the ACM SIGMOD International Conference on Management of Data. 1979.

45. Shannon, C.E. A Mathematical Theory of Communication, Bell Syst. Tech. Journal 27, 379-423, 623-656. 1948.

46. Shasha, D., Wang, J. T. L., Giugno, R. Algorithmics and Applications of Tree and Graph Searching. In PODS, 2002

47. Spiegel, J., Polyzotis, N.: Graph-Based Synopses for Relationa Selectivity Estimation. In Proc. 2006 ACM SIGMOD Conference, 2006.

48. Srivastava, J., Cooley, R., Deshpande, M., Tan, P. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. ACM SIGKDD Explorations 2000.

49. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In WWW 2008.

50. SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes.

51. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking Database Representations of RDF/S Stores. In Proc. 4th ISWC Conference, 2005.

52. Wang, C., Parthasarathy, S., Jin, R.: A Decomposition-Based Probabilistic Framework for Estimating the Selectivity of XML Twig Queries. In EDBT, 2006.

53. Wang, C., Wang, W., Pei, J., Zhu, Y., Shi, B. Scalable Mining of Large Disk-based Graph Databases. KDD 2004.

54. Wang, W., Jiang, h., Lu, H., Yu, j.: Bloom Histograms: Path Selectivity Estimation for XML Data with Updates. In Proc. 30th VLDB Conference, 2004.

55. Wilkinson, K.: Jena Property Table. In Proc. 3rd Jena User Conference, 2006.

56. Winer, F. B. Drug design and the CAS ONLINE Substructure Search System. Drug Inf J. 983;17(4):277-86.

57. Wu, J., Patel, J., Jagadish, H. V.: Estimating Answer Sizes for XML Queries. In Proc. 8th EDBT Conference, 2002.

58. Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In Proc. 2nd ICDM Conference, 2002.

59. Yan, X., Yu, P. S., Han, J. Graph Indexing: A Frequent Structure-based Approach. In SIGMOD, 2004.

60. Zhao, P., Yu, J. X., Yu, P. S.: Graph Indexing: Tree + Delta >= Graph. In VLDB, 2007

61. Oracle® Spatial Resource Description Framework (RDF) 10*g* Release 2 (10.2) Manual

Appendix A. **Queries in SPARQL**

Query 1

For each country, find its provinces, their capitals and the rivers that flow through the country

```
Prefix protégé: <http://protege.stanford.edu/kb#>
Select ?country ?province ?province_capital ?river
Where
{
?country protégé:has-province ?province.
?province protégé:has-province-capital ?province_capital.
?river protégé:flows-through-country ?country
}
```

Query 2

For each country, find its provinces, their capitals and the languages of the country

```
Prefix protégé: <http://protege.stanford.edu/kb#>
Select ?country ?province ?province_capital ?language
Where
{
?country protégé:has-province ?province.
?province protégé:has-province-capital ?province_capital.
?country protégé:languages ?language
}
```

Query 3

For each country that is a member of the World Trade Organization (WTrO), find its provinces and their capitals

```
Prefix protégé: <http://protege.stanford.edu/kb#>
Select ?country ?province ?province_capital
Where
{
?country protégé:has-province ?province.
?province protégé:has-province-capital ?province_capital.
protégé:WTrO  protégé:member ?country
}
```

Query 4

For each country that is in Africa, find its provinces and their capitals

```
Prefix protégé: <http://protege.stanford.edu/kb#>
Select ?country ?province ? province_capital ?continent
Where
{
?country protégé:has-province ?province.
?province protégé:has-province-capital ?province_capital.
?country protégé:encompassed ?continent.
?continent protégé:name “Africa”
}
```

Query 5

For each professor that directs a research group, find the course he teaches and the TA of the course

```
Prefix lubm: <univ-bench.owl#>
Select ?researchGroup ?professor ?course ?ta
Where
{
?researchGroup lubm:director ?professor.
?professor lubm:teacherOf ?course.
?ta lubm:teachingAssistantOf ?course
}
```

Query 6

For each professor that is a co-PI of two projects, find the research group of the projects

```
Prefix lubm: <univ-bench.owl#>
Select ?project1 ?project2 ?professor ?researchGroup1 ?researchGroup2
Where
{
?project1 lubm:coPI ?professor.
?project2 lubm:coPI ?professor.
?researchGroup1 lubm:hasProject ?project1
?researchGroup2 lubm:hasProject ?project2
}
```

Query 7

For each project that has two sub projects, find the publications of the project authored by FullProfessor0 in department10, university0, and the citations of the publication

```
Prefix lubm: <univ-bench.owl#>
Prefix journal1: <http://www.Journal1.org/>
Select ?professor ?department ?publication
Where
{
?professor lubm:headOf ?department.
?publication lubm:publicationAuthor ?professor.
?publication lubm:publicationInJournal journal1:University0/Department0
}
```

Query 8

Find the journal publications of all head of departments whose name is FullProfessor2.

```
Prefix lubm: <univ-bench.owl#>
Select ?professor ?department ?publication ?journal
Where
{
?professor lubm:headOf ?department.
?professor lubm:name “FullProfessor2”.
?publication lubm:publicationAuthor ?professor.
?publication lubm:publicationInJournal ?journal
}
```

Appendix B. **Combining the Preference and Estimation Values**.

We now show how the preference and estimation values of patterns are combined to obtain a single value with which the P-Tree is pruned.

Definition 16. Let $P = \{p_1, p_2, \ldots, p_m\}$ be the set of patterns in the P-Tree and p_{EVmax}, the maximum expected value of patterns in P. Given a constant $c > 0$, the value of a pattern p_j is given by:

$$p_{Vj} = (1 + p_{EVj})(1 + p_{PVj}) + i p_{EVmax}$$

where i is an indicator variable whose value is 1 if $p_{PVj} \geq c$ and 0 otherwise. The additive constants ensure that the value of a pattern is non-zero when either its preference or estimation value is zero. On the other hand, the second term allows for tuning the P-Tree by boosting the values of important patterns as defined by their preference values, to delay their pruning.

We note that when the size of the P-Tree exceeds the budget, we prune the tree systematically to meet the budget. To prune, we compute the value of each node of the P-Tree (the combination of its preference and estimation values, or simply the latter if the former is not given) and select nodes to be pruned as shown in the *Prune P-Tree* algorithm shown below.

```
Prune P-Tree
Input:  P-Tree T, Budget B, constant c
Output: Pruned P-Tree T
1.  δ ← 0; inc ← 0; Set estimable ← ∅; done ← false
2.  while done = false do
3.        estimable ← ∅
4.        for each internal node v in T in bottom – up order do
5.              compute estimation value given δ
6.              if v's estimation value > 0 then
7.                    insert all its children into estimable
8.              end if
9.        end for
10.       if sizeof(T) - sizeof(estimable) ≤ B and δ is optimal then
11.             compute values of patterns in T
12.             if c > 0
13.                   delete smaller valued patterns to meet B
14.             else
15.                   prune all patterns in estimable
16.             end if
17.             done ← true
18.       else
19.             adjust δ
20.       end if
21. end while
```

As lines 4-9 of the Prune-Tree algorithm show, estimation values are used to select the patterns to be pruned. However in lines 12-16, the patterns eventually pruned may differ when tuning the tree. The running time of the algorithm $O(Ld^{maxL}\log(\max_i\{freq(p_i)\}))$, is reasonable since logarithm is a slowly growing function and maxL will typically be small. The running time stems from the loops in lines 4-9 and 2-21. Lines 4-9 run in $O(Ld^{maxL})$ time, where L and d are the numbers of unique edge labels and the maximum degree of nodes in the graph respectively. Recall that the root of the P-Tree has a child for each unique edge label in the graph while internal nodes have at most d children. Lines 2-21 will be executed at most $\log(\max_i\{freq(P_i)\})$ times, since all patterns are estimable at $\delta = \max_i\{freq(P_i)\}$

Appendix C. **Installation Instructions.**

The software is currently supported only on Linux platforms. This software uses Brahms so the user has to download and install Brahms first. To download and install Brahms, follow the instructions found at http://lsdis.cs.uga.edu/projects/semdis/brahms/. After Brahms has been downloaded and installed, install this software by unzipping and untarring the file GraphSummaries.tar.gz. Then go to the source directory in the GraphSummaries distribution. There are two phases to using the software. The first phase creates the graph summaries whereas the second phase uses the created summaries to guide query optimizers in choosing an optimal plan for query processing. To create a graph summary, use the "GeneratePatterns" binary file. There are several input parameters to the "GeneratePattern" binary file. These can be viewed by running GeneratePatterns with the help option. After the summary has been created, they can be used by calling the "processQuery" method in "QueryProcessor". This method accepts several input parameters. They are:

const char* modelName: the name of the snapshot file created using the snapshotCreator utility distributed with Brahms.

string queryFileName: The name of the file that contains the formatted query. The query is formatted as a sequence of triples of the form *subject predicate object* where the subjects and objects are preceded by the integer 0, 1, 2, or 3 to denote that they represent a literal, a variable, a uri or a class respectively. See the file "queryFormatDescription" for more details and examples.

string summaryFileName: The name of the summary file created in the first phase to be used for query procfessing.

string lookUpFileName: The file name of the look up table that was created in the first phase to be used for query processing.

string litLookUpFileName: The file name of the literal look up table that was also created in the first phase to be used for query processing.

usint summaryType: An unsigned short integer that denotes the type of graph summary that is being used either 2 for Pattern Tree or 3 for Maximal Dependence Tree.

usint maxSize: An unsigned short integer that indicates the maximum size/length of patterns in the graph summary being used.

bool default: Indicates if default values are kept for each pattern size for estimation purposes.

int evalType: An integer denoting the type of evaluation to be done in choosing the optimal plan. It could be 0, 1 or 2 for cost, cardinality or selectivity respectively.

int enumType: An integer denoting the type of enumeration for the iterative dynamic programming plan enumeration strategy. It could be 0 for standard or 1 for balanced.

usint idpStep: An integer that specifies the number of dynamic programming steps to perform before the greedy selection, in the iterative dynamic programming plan enumeration strategy.

int joinType: An integer that specifies the type of join algorithm to be used. It could 0 for nested loop join, 1 for hash join or 2 for merge join.

bool elimDup: Specifies if duplicates should be eliminated

int planEnumType: An integer that specifies which plan enumeration strategy to be used. It could be 0 for dynamic programming enumeration, 1 for greedy enumeration, 2 for iterative dynamic programming enumeration and 3 for using the SPARQLeR query planning technique.

int estimationType: An integer that specifies the type of estimation to be done. It could be 0 for the estimation using the proposed graph summaries or 1 for using the dbms estimation style.

int sip: An integer that specifies how execution should proceed. It could be 0 for enumerating all plans with intermediate table generation, 1 for enumerating only left-deep plans with iterator/piped executions, or 2 for enumerating only left deep plans with intermediate table generation.

double selectivityDecay: A double value that specifies how the propagation of the selectivities of constrained join queries to larger sized patterns should be decayed.

int numRepetitions: An integer that specifies the number of times a query should be repeated for average timings.

The experiments conducted in this work can be reproduced by using the datasets used in this work. The Mondial dataset is freely available on the web at http://www.informatik.uni-ulm.de/ki/Liebig/owl/mondial.owl. We converted this owl file to RDF by using Protégé. The LUBM dataset is freely available at http://swat.cse.lehigh.edu/projects/lubm/. However, like we said, we modified this LUBM dataset generator by adding more properties and classes. This modified version is included with the distribution of this software.