On Building an Index Advisor for Semantic Web Queries

Lubomir Stanchev [a] Grant Weddell [b]
[a] *Indiana University - Purdue University Fort Wayne, USA*
[b] *University of Waterloo, Canada*

Abstract. Current optimization techniques for answering queries over Semantic Web data use realization to precalculate the individuals associated with every concept in the given ontology. However, this technique does not take into account the type of queries, written for example in nRQL or SPARQL-DL, that will arrive at the system. In this paper we propose how this additional knowledge can be used to create query-specific indices. We include experimental results that show how our approach can be used to improve the performance of the Pellet query engine for the popular LUBM benchmark.

1. Introduction

The Semantic Web and related terms have become buzz phrases in the last few years. For example, John Markoff coined the phrase Web 3.0 in the New York Times in 2006 referring to providing semantic capabilities that enhance the user experience. Although the World Wild Web Consortium (W3C) has attempted to standardize related languages (OWL, RDF, and SPARQL, to name a few), we are still far from realizing the dream of the Semantic Web because current algorithms for retrieving information from knowledgebases do not scale well for large data sources, such as the current size of the Internet. In this paper we try to address some of these scalability challenges by describing how search principles from relational database technology can be applied to knowledgebases. Specifically, relational database engines use index advisors to select indices that can optimize query workloads consisting of SQL queries. Here, we adopt a similar approach and describe how an index advisor can be used to construct *description indices* that can be used to optimize KIF-like queries ([1]) over an OWL-DL knowledgebase.

This is an important problem because without techniques for efficiently accessing knowledgebases their use will remain isolated to narrow domains, such as medical terms (e.g., the SNOMED ontology [2]). In most cases, the type of queries that will be asked from a knowledgebase are known upfront up to some parameters and therefore it is reasonable to create search data structures that can be used to efficiently answer the queries.

The problem that is addressed in this paper is intrinsically hard because conventional indexing techniques from relational databases fall short when it comes to querying knowledgebases. For example, consider a knowledgebase of chairs. For some chairs we may know that they are from the Medieval Period, while for other chairs we may know that they are from the European Renaissance, and there may be even chairs from the eighteenth century. Organizing these chairs into an index that is ordered relative to age is not a trivial task because additional knowledge of how these periods are related to each other is needed. Therefore, calls to a concept subsumption reasoner that can reason with additional data (called terminology) needs to be used when constructing or traversing an index.

Current solutions for solving the problem of indexing knowledgebases use realization to precompute the elements of each concept (see for example the Racer approach [3]). However, this is done independently of the type of input queries. For example, consider the following query, which is a modified version of a query from the LUBM benchmark ([4]), expressed in a KIF-like language.

(type Professor ?X) (type Department :P) (worksfor ?X :P) (order ?X by rank)

The query asks for all professors that work for a given department ordered by their rank. We have used :P to denote a parameter to the query. In order to answer this query efficiently, we need to know all the professors in each department. The professors in a department need to be ordered relative to their subtype (e.g., full professors come first, followed by associate and assistant professors). An instance of the novel *description index* that is described in this paper will store the concept descriptions associated with every department individual. For each such individual, the index will also store the related professor individuals in the described order. Such a description index can be used to efficiently answer the example query. In contrast, such efficient access is not possible with existing techniques implemented in knowledgebase query engines such as Pellet ([5]) and Racer ([6]). The reasons are that: (1) existing engines do not create query specific indices and (2) they do not explore novel index types that are required for supporting knowledgebase queries.

On a related note, note that our approach is very different than the approaches taken by RDF databases, such as RDF3X ([7]), Hexastore ([8]) and LUPOSDATE ([9]). While these systems index RDF triplets, our approach indexes concept descriptions. As a result, a description index can store much richer information than an index on RDF triplets.

Our approach first examines each input query and considers possible plans for executing it (see for example [3]). Although different query plans can examine, for example, different join orders, they will have the same primitive queries as the building blocks. Efficient execution of these primitive queries is required for the efficient execution of the overall query. In this paper we consider primitive queries that have certain syntax. Such queries can be executed in the relational case in logarithmic time. When the input data is a knowledgebase, the retrieval time can be worse and will depend on the amount of uncertainty that is introduced in the data. Our final step is to create a description index for each primitive query. Note that the merging of description indices should also be investigated because it can potentially reduce the storage overhead without affecting the execution time of the input queries. However, index merging is beyond the scope of this paper and is left as a topic for future research.

One known limitation to our approach is that it relies on a query optimizer to break complex queries into simpler ones. However, we believe that this is unavoidable because an index advisor should be custom-fitted for the query optimizer that will use it (i.e., an index is only good if the query optimizer will use it). Another drawback of our approach is that indices with replicated data are created when the concept descriptions that describe the individuals in the knowledge are not descriptive enough (e.g., a person is defined as being older than twenty years of age). This can adversely affect both search and update time. One possible way to overcome this limitation is to consider index compression that will remove redundant copies of the same data. Similar to index merging, this research area is outside the scope of the paper and a topic for future investigation.

The main contributions of the paper are the new description index type that are described in Section 3 and the description index creation procedure that is presented in Section 4. Section 2 of the paper gives an overview of related research, while Section 5 outlines the results of our experimental evaluation.

2. Related Research

The Web Ontology Language - Description Logics (OWL-DL) [10] is introduced by the World Wild Web Consortium (W3C) as a standard for representing knowledgebases. While most existing research focuses on consistency checking using a tableau-based approach (Pellet [5], Fact++ [11], and Racer [6] are three popular implementations), there have been two notable proposal for SQL-like query languages over OWL-DL knowledgebases.

The first one is SPARQL-DL ([12]). The language is very expressive and the free variables can be not only individuals, but also concepts or relationships (i.e., the languages has capabilities that cannot be expressed in first-order logics). Notably, [13] describes how cost-best query optimization can be used to make query execution decisions, such as join-order selection. Currently, SPARQL-DL is supported by Pellet. The second language is nRQL and was created by the developers of Racer [14].

Papers, such as [3], explain how to use realization to precompute the elements of each concept in the terminology. Although this approach reduces the number of expensive calls to the reasoner, it can still leads to linear time complexity for answering certain queries, such as our example query. The reason is the lack of appropriate indices.

Ground breaking work in the area includes the papers [15,16]. They describe how to create an index that stores concept descriptions rather than individuals. The concept descriptions in the index are ordered relative to inferred values for some of the data properties. In this work we extend this research by describing how to create more complex indices that can support efficient execution of nested queries. (The two papers only address the problem of finding all the individuals that belong to a given concept description.)

3. Description Indices

In this section we introduce the concepts: *description tree index* and *description hash table*, which are the building blocks of a description index. The two artifacts correspond to search tree index and hash table, respectively, in the relational database sense.

A description tree index stores concept descriptions in a predefined order and allows for efficient range search and for retrieving the result relative to a predefined order. A description hash table stores concept descriptions that are clustered relative to inferred values of one or more properties and supports efficient answering of partial-match queries. We assume that the concept description are written in OWL-DL using the Manchester

(*concept*)	(*subsumed by*)
Full Professor	Professor **and** (*works for* **some** Organization)
Graduate Student	Student **and** (*advisor* **only** Professor)
Undergraduate Student	Student **and** (*takes course* **min** 2) **and** (*takes course* **max** 4)
Student	Undergraduate Student **or** Graduate Student

Table 1. Part of the LUBM Schema Expressed Using the Manchester OWL Syntax

OWL Syntax([17]). For completeness, we next present example expressions in this language, where the reader should refer to [10,17] for a complete overview.

3.1. The Manchester OWL Syntax

Our running example is based on the terminology of the LUBM benchmark. Part of the terminology, expressed using the Manchester OWL Syntax, is shown in Table 1, where bold is used for language reserved words, while properties (both data and object) are in italic.

The first subsumption states that if someone is a full professor, then they are a professor and they work for at least one organization. The second rule states that if someone is a graduate student, then they are a student and all their advisors are professors. The third rule states that undergraduate students must take between two and four courses. The last rule states that if someone is a student, then they must be either an undergraduate or graduate student.

Note that the first column of the table contains primitive concepts, while the second column contains *concept descriptions* (a primite concept is a special case of a concept description). A concept description describes all individuals that satisfy certain rules. For example, the concept description in the right column of the first row in Table 1 describes all professors that work for an organization. The relationship between the two columns of the table is *subsumption* (denoted as **SubClassOf**), that is, if an individual belongs to the primitive concept shown in the left column, then it must also belong to the set of individuals described by the concept description in the right column.

A set of subsumption rules form a *terminology* (i.e., the additional knowledge that is used when answering queries). The Manchester OWL Syntax is also used to describe the *individuals* in the knowledgebase. For example, a particular student may be described as follows.

is taking Calculus 101 **and** *is taking* Stats 101 **and** *member of* Computer Science Department **and** *age* **value** $[>= 20]$

This student takes Calculus 101 and Statistics 101, they are member of the Computer Science Department, and they are at least twenty years of age.

3.2. Description Tree Index

A formal definition of a description tree index follows.

Definition 1 (description tree index) *A description tree index has the syntax* $\langle C, O\rangle$*, where* C *is a concept description and* O *is an ordering description. The index will store* *all individuals e that have concept description D for which it can be inferred from the terminology $\mathcal{T}$ that C subsumes D (i.e. $\mathcal{T}$ implies D* ***SubClassOf*** *C). The ordering description O defines the order of the elements in the search tree.*

The definition relies on a way for defining ordering among concept descriptions, which is presented next.

Definition 2 (ordering description) *An ordering description Od has the following syntax.*

$Od ::= \langle\rangle \mid \langle f\ \mathtt{dir} : Od\rangle \mid \langle D_1 : Od_1, \ldots, D_m : Od_m\rangle$

We have used $\langle\rangle$ to denote an empty ordering, `dir` *to denote* `asc` *or* `desc`*, D to denote a concept description, and f to denote a data property. In order for this ordering description to be valid relative to a terminology, it must be the case that "D_i* **and** *D_j* **SubClassOf EMPTY***" for $1 \leq i \neq j \leq m$ and "D* **SubClassOf** *D_1* **or** *...* **or** *D_m" are both implied by the terminology, where D is a concept description that describes the set of individuals on which the order is defined (***EMPTY*** *is used to denote the empty concept - i.e., bottom in description logics). We will say that the concept description D is before E relative to an ordering description Od and terminology $\mathcal{T}$ and write $D \prec_{Od,\mathcal{T}} E$ in the following cases (x^* is used to denote the renaming of x to x^*):*

- *when Od is $\langle f\ \mathtt{dir} : Od_1\rangle$:*
  - $*$ *$\mathcal{T} \cup \mathcal{T}^*$ implies $(D$* **and** *$E^*)$* **SubClassOf** *$(f < f^*)$ when* `dir=asc` *and $(D$* **and** *$E^*)$* **SubClassOf** *$(f > f^*)$ when* `dir=desc` *or*
  - $*$ *$\mathcal{T} \cup \mathcal{T}^*$ implies $((D$* **and** *$E^*)$* **SubClassOf** *$(f = f^*))$* **and** *$D \prec_{Od_1,\mathcal{T}} E$*
- *when Od is $\langle D_1 : Od_1, \ldots, D_m : Od_m\rangle$:*
  - $*$ *$\mathcal{T}$ implies $(D$* **SubClassOf** *$D_i)$* **and** *$(E$* **SubClassOf** *$D_j)$ for some $i < j$ or*
  - $*$ *$\mathcal{T}$ implies $(D$* **or** *$E)$* **SubClassOf** *D_i for some i and $D \prec_{Od_i,\mathcal{T}} E$.*

Note that we may omit some of the angle brackets in an ordering description when clear from the context. Similarly, when `dir` is `asc`, it may be omitted. Note that if f and g are data properties, then $f = g$ ($f > g$) represents the set of individuals that have the same value for the two properties (for which the value for the f property is bigger than the value for the g property).

An example of a description tree index is: $\langle$Professor, $\langle$*worksFor.name*: $\langle$Full Professor : *age*, Associate Professor : *salary*, Assistant Professor: *publications*$\rangle\rangle\rangle$. It denotes a search tree that contains all individuals that can be inferred to be professors. The ordering will be first relative to the name of the place of employment. In the same workplace, full professors will come first, followed by associate and assistant professors. Full professors in the same workplace will be ordered relative to age, associate professors - relative to salary, and assistant professors - relative to the number of publications.

Consider a query that asks for all associated professors with a given name (we will denote this name as :P) ordered by salary. To answer the query, we can do a search in the description tree index, where for simplicity we assume that the search tree is binary. For every node with concept description D we need to compare the concept descriptions D and "Associate Professor `and` $name =$:P" relative to the ordering description of the description tree index. If D comes first, then we need to continue the search in the right subtree. If D comes second, then we need to continue the search in the left subtree. If the two concept descriptions are not comparable relative to the ordering description, then we need to check if the terminology implies "D **SubClassOf** (Associate Professor $\texttt{and}$ $name$ = :P)". If this is the case, then we have found a concept description that belongs to the query result and we know that the other concept descriptions that belong to the result will be adjacent relative to an in-order traversal of tree. If the two concept descriptions are not comparable and the terminology does not imply this subsumption, then we need to check both the left and right subtrees of the current node. The reason is that in general an ordering description defines a partial order.

If we need the search to be always efficient (i.e., return each element of the query result in logarithmic time), then we need to enforce the following *descriptive sufficiency* property ([15]) for all concept descriptions in the description tree index. Note that the descriptive sufficiency property holds for the data that is generated from the LUBM benchmark.

Definition 3 (descriptive sufficiency) *A concept description D is sufficiently descriptive relative to an ordering description Od and terminology $\mathcal{T}$, written as $DS_{Od,\mathcal{T}}(D)$, if one of the following holds.*

- $Od = \langle\rangle$
- $Od = \langle f \; \texttt{dir} : Od_1\rangle$, *$DS_{Od_1,\mathcal{T}}(D)$, and $\mathcal{T}$ implies D* **SubClassOf** *$f = k$ for some k*
- $Od = \langle D_1 : Od_1, \ldots, D_m : Od_m\rangle$, *$DS_{Od_i,\mathcal{T}}(D)$, and $\mathcal{T}$ implies "D* **SubClassOf** *D_i" for some i*

Next, consider a node in a description tree index with concept description C_1 that is "*age* **value** $[1, 8]$" and a left child node with concept description C_2 that is "*age* **value** $[5, 7]$". Suppose that the concepts in the tree are ordered relative to the ordering description $\langle age\rangle$. Now, if a new concept description C_3 with value "*age* **value** 3" is inserted in the knowledgebase, then it can be inserted either in the left or right child subtree of the root node because C_1 and C_3 are not comparable relative to the defined search tree order. However, inserting C_3 in the right subtree is obviously the wrong choice because $C_3 \prec_{\langle age\rangle,\mathcal{T}} C_2$. Therefore, when inserting a new element in a concept description index, we may need to consider both the left and right subtree of the current node if the descriptive sufficiency property does not hold for the indexed descriptions.

3.3. Description Hash Table

The general definition of a description hash table follows.

Definition 4 (description hash table) *A description hash table H has the syntax $\langle C, \{p_i\}_{i=1}^{k}\rangle$, where C is a concept description and $\{p\}_{i=1}^{k}$ are properties. It partitions all concepts descriptions that are subsumed by C using a hash function on the different values of $\{p\}_{i=1}^{k}$.*

Note that the second element of the syntax of a description hash table may be empty, in which case we will store all concept descriptions in a single bucket, that is, the description hash table will degenerate into a *description list*.

An example description hash table is: $\langle$Graduate Student `or` Undergraduate Student, $\{advisor\}\rangle$. It partitions the graduate and undergraduate students relative to their advisor. If the advisor of a student is not known, than this student could be placed in several hash buckets. For example, if the advisor of a student that is in the Computer Science Department is not known, but we know that students in the Computer Science department are supervised by professor from the Computer Science Department, then we only need to store the student in the hash buckets for Computer Science professors.

Searching in a description hash table is similar to searching in a regular hash table in the sense that the clustering allows us to prune buckets that do not contribute the query result. Since the same concept description can appear in multiple hash buckets, response time for search can degrade if the concept descriptions that are stored in the index are not descriptive.

3.4. Description Index

The definition of a description index follows.

Definition 5 (description index) *A description index can be represented as a rooted tree, which we will call the* definition tree *of the description index. The nodes in the tree can be either* hash nodes *or* tree nodes. *The label of a hash node is that of a description hash table, while the label of a tree node is that of a description tree index. The edges in the definition tree have labels that are object properties.*

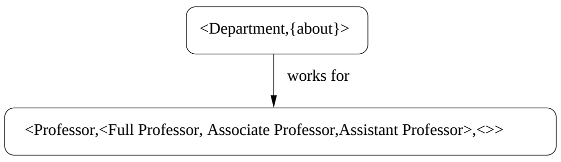
A description index represents a set of description hash tables and description tree indices, where the connection between the data structures and their content is determined by the shape and labels of the definition tree.

Specifically, for a root node in the definition tree that is a hash node or a tree node, the corresponding description hash table or description tree index, respectively, is built. If the node n_1 is the child of the node n_2 and the edge between the two nodes has the label p, then a data structure is built for each concept description D stored in the data structure for n_2. Specifically, if n_1 is the tree node $\langle C, O\rangle$, then the description tree index $\langle C$ `and` $(p$ `some` $D), O\rangle$ is created. Alternatively, if n_1 is the hash node $\langle C, \{p_i\}_{i=1}^{k}\rangle$, then the description hash table $\langle C$ `and` $(p$ `some` $D), \{p_i\}_{i=1}^{k}\rangle$ is created.

An example description index is shown in Figure 1. Note that we have used *about* to denote the identifier of an element (it is represented as *rdf:about* in the LUBM benchmark data). The index contains a description hash table that stores all elements that can be inferred to describe a department. For each such element, a description tree index that stores all elements that can be inferred to be professors in the respective department is created, where the professors are ordered relative to their rank. Now, our example query from Section 1 can be answered by first finding the department with the specified value for *about* and then finding all the professors in this department using the corresponding description tree index.

4. Index Creation

In this section we explain how description indices are created from primitive queries.



<Department,{about}>
works for
<Professor,<Full Professor, Associate Professor,Assistant Professor>,<>>

Figure 1. Definition tree of an example description index

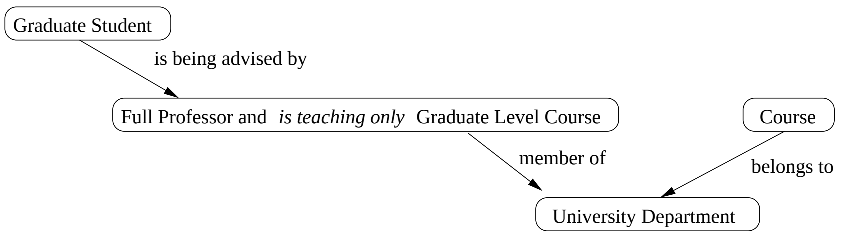


Figure 2. Example query graph

Definition 6 (primitive queries) *Primitive queries have a query graph that is in the shape of an inverse tree. Each node is labeled with a concept description, while edges are labeled with object properties. The query represented by the graph is a conjunction of KIF-like terms. For each node, the term "(type D_i ?x_i)" is constructed, where D_i is the label of the node. For each edge between D_i and D_j labeled p_q the term "(p_q ?x_i ?x_j)" will be created. In addition, KIF-like terms involving parameters can be specified on the concept description for the root node. This terms can be of form "(p_1 value :P_1)* **and** *...* **and** *(p_k value :P_k)* **and** p_{k+1} `value` $[>=: P_{k+1}, <= P_{k+2}]$*", where the partial match terms or the range term may be missing. We postpone specifying the restriction on the ordering condition of the query.*

An example primitive query in this format is: "(**type** Graduate Student $?x_1$) (**type** Full Professor **and** *is teaching* **only** Graduate Level Course $?x_2$) (*is being advised by* $?x_1$ $?x_2$) (**type** University Department $?x_3$) (*member of* $?x_2$ $?x_3$) (*name* ?x3 :P1) (**type** Course $?x_4$) (*belongs to* $?x_4$ $?x_3$)", where the query graph is shown in Figure 2. The query asks for information about all graduate students who are advised by full professors that are members of a given department and who teach only graduate level courses. The query also asks about the courses in that department. The query will return information about the student, their advisor, the department of their advisor, and the all the courses in the department.

Given a primitive query that follows the described syntax, we create a description tree that has a definition tree that is the inverse tree of the query graph. In our example, the node for the university department will be the root node of the definition tree and the nodes for course and full professor will be its direct children. Note that the labels for the edges do not change, while the labels for the nodes change as described in the next paragraphs.

We will first explain how to construct the labels for the leaf nodes in the inverted tree of the query graph (the graduate student node in our example). If the leaf node is based on the concept D_1, then the node will be a hash node with label $\langle D, \{\}\rangle$ if no ordering is defined on the individuals that are subsumed by D. Alternatively, if an ordering condition is specified, then a tree node with concept description D and the ordering is created. For our example query, the hash node $\langle$Graduate Student,$\{\}\rangle$ will be created. Alternatively, if the query asked for the graduate students of the same advisor to be ordered by age, then the tree node $\langle$Graduate Student,$\{$*name*$\}\rangle$ will be created.

We next explain how the labels for the non-leaf and non-root nodes are created. If a node n has label D and no ordering condition is expressed on it, then the hash node $\langle D$ **and** $(p_1$ **some** $D_1)$ **and** $\ldots$ $(p_k$ **some** $D_k), \{\}\rangle$ is created. In the expression, $\{D_i\}_{i=1}^k$ are concept descriptions for the child nodes in the definition tree that have already been constructed and $\{p_i\}_{i=1}^k$ are the respective inverse properties of the edges that connect the node to its children. In our example query, for the professor node will create the hash node $\langle$Full Professor **and** *is teaching* **only** Graduate Level Course **and** *advises* **some** Graduate Student,$\{\}\rangle$ because no ordering is defined on professors.

The label for the root node is defined similarly, with the addition that the parameterized predicate, when present, needs to be taken into account. Specifically, if the constraint "$(p_1$ value :$P_1)$ **and** $\ldots$ **and** $(p_k$ value :$P_k)$ **and** p_{k+1} `value` $[>=: P_{k+1}, <= P_{k+2}]$" is defined, then the tree node $\langle \ldots, \langle p_1, \ldots, p_{k+1}\rangle\rangle$ will be created. An ordering condition can also be specified in the query, where p_{k+1}, when a range predicate is defined, should be the first property in the ordering condition. If both an ordering condition and a range predicate are missing, then a hash node will be created. In our example, the hash node "$\langle$University Department **and** *offers* **some** Course **and** *has as member* **some** (Full Professor **and** *is teaching* **only** Graduate Level Course **and** *advises* **some** Graduate Student), $\{$*name*$\}\rangle$" will be created.

We require that the ordering of the query is such that it follows the nodes of the definition tree of the description index. More precisely, the properties for each concept description must appear consecutively in the ordering and in the order of the definition tree. In our example, this means that the properties for the department should come first, potentially followed by properties of professor and graduate student, where the properties for the course can come any time after the properties for the department. Finally, if the ordering for a node with concept description D is not total, then there should be no ordering defined on the descendants of that node in the definition tree. In our example, $\langle$Department.*name*, Professor.*salary*$\rangle$ is a valid ordering as long as there are no two departments with the same name.

The so constructed description index can be used to answer the input query. The traversal will be efficient as long as the stored concept descriptions are sufficiently descriptive and all edges in the query graph are labeled with functional properties.

5. Experimental Results

Our experimental results are based on the LUBM benchmark([4]). We created a database that consists of twenty universities. The benchmark contains a program that generated data about each university (e.g., departments, students, professors, courses, etc.). The benchmark consists of fourteen queries. We executed each of the queries in two different modes. One way is using the description index structures that are described in the paper, were the indices were manually custom-built using our algorithm. The other way is by rewriting the queries in SQARQL-DL and executing them through Jena interface and using the Pellet reasoner. In both cases the heap was set to 1.5GB to avoid an out of

(*query*)	(*description index*)	(*Pellet*)
Q1	1	1076
Q2	47	3167
Q3	1	1263
Q4	1	1373
Q5	1	796
Q6	671	3619
Q7	1	1373
Q8	35	1966
Q9	51	4712
Q10	520	1404
Q11	31	1186
Q12	1	3213
Q13	35	858
Q14	468	2293

Table 2. Execution time for the LUBM queries in milliseconds

(*query*)	(*index size*)
Q1	5
Q2	4
Q3	3
Q4	3
Q5	14
Q6	13
Q7	14
Q8	21
Q9	15
Q10	14
Q11	1
Q12	1
Q13	14
Q14	10

Table 3. Size of indices for each query in MBs

space error. Both tests were executed on Sony VGN-NW150J laptop with 4GB of main memory and 2.10GHz CPU. The results of the experiments are shown in Table 2. We measured the time to compute and print the result, where the time to initially load the data was not measured. For example, Query 6 runs slowly in both scenarios because it asks for all the students and printing all students is time consuming. Conversely, Query 2 asks for all graduate students that are students in the university from which they got their undergraduate degree. Our solution performs an index join, which is significantly faster than the nested-loop join that is performed by Pellet.

We realize that there is a cost to pay for creating indexes. The obvious cost is the cost of storage. The size of the initial knowledgebase of twenty universities is 110MB. The amount of storage needed to store the indices for each query is shown in Table 3. As expected, indices bring slight storage overhead.

6. Conclusion

In this paper we described how to create description indices that can be used to index semantic web data. We believe the paper builds the foundation for creating an index advisor for SPARQL-DL queries over knowledgebases. Areas for future research include integrating the algorithm that is proposed in the paper in existing semantic query optimizers, extending description indices, and considering different description index merging strategies.

References

[1] Genesereth, M.R., Fikes, R.E.: Knowledge Interchange Format Version 3.0 Reference Manual. Standord Technical Manual http://www.ksl.stanford.edu/knowledge-sharing/kif/.

[2] : International Health Terminology Standards Development Organization. http://www.ihtsdo.org/

[3] Haarslev, V., Möller, R.: Optimization Techniques for Retrieving Resources Described in OWL/RDF Documents: First Results (2004). In Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR) (2004)

[4] Pan, Z., Guo, Y., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2) (2005) 158–182

[5] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2) (2007)

[6] Haarslev, V., Möller, R.: Racer: A Core Inference Engine for the Semantic Web. Second International Workshop on Evaluation of Ontology-based Tools (2003)

[7] : RDF-3X. http://www.w3.org/RDF

[8] Weiss, C., Karras, P., Bernstein, A.: Hexastore:Sextuple Indexing for Semantic Web Data Management. VLDB2008

[9] Groppe, S., Groppe, J.: External Sorting for Index Construction of Large Semantic Web Databases. Proceedings of the 25th ACM Symposium on Applied Computing (ACM SAC 2010)

[10] : OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/

[11] Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. Lecture Notes in Computer Science **4130** (2006) 292–297

[12] Sirin, E., Parsia, B.: SPARQL-DL: SPARQL Query for OWL-DL. 3rd OWL: Experiences and Directions Workshop (OWLED) (2007)

[13] Sirin, E., Parsia, B.: Optimizations for answering conjunctive abox queries: First results. In Proceeding of the International Description Logics Workshop (2006)

[14] Haarslev, V., Möller, R., Wessel, M.: Querying the semantic web with Racer + nRQL. In Proceedings of the KI-2004 International Workshop on Applications of Description Logics (2004)

[15] Pound, J., Stanchev, L., Toman, D., Weddell, G.E.: On ordering descriptions in a description logic. International Workshop on Description Logics (2007)

[16] Pound, J., Stanchev, L., Toman, D., Weddell, G.E.: On Ordering and Indexing Metadata for the Semantic Web. International Workshop on Description Logics (2008)

[17] Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.: The Manchester OWL syntax. OWL: Experiences and Directions (2006)