A Node Indexing Scheme for Web Entity Retrieval

Renaud Delbru[1], Nickolai Toupikov[1], Michele Catasta[2⋆], and Giovanni Tummarello[1,3]

[1] Digital Enterprise Research Institute
National University of Ireland, Galway
Galway, Ireland
firstname.lastname@deri.org
[2] School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
firstname.lastname@epfl.ch
[3] Fondazione Bruno Kessler
Trento, Italy
lastname@fbk.eu

Abstract. Now motivated also by the partial support of major search engines, hundreds of millions of documents are being published on the web embedding semi-structured data in RDF, RDFa and Microformats. This scenario calls for novel information search systems which provide effective means of retrieving relevant semi-structured information. In this paper, we present an "entity retrieval system" designed to provide entity search capabilities over datasets as large as the entire Web of Data. Our system supports full-text search, semi-structural queries and top-k query results while exhibiting a concise index and efficient incremental updates. We advocate the use of a node indexing scheme and show that it offers a good compromise between query expressiveness, query processing time and update complexity in comparison to three other indexing techniques. We then demonstrate how such system can effectively answer queries over 10 billion triples on a single commodity machine.

1 Introduction

On the Web, more and more structured and semi-structured data sources are becoming available encouraged by initiatives such as Linked Open Data, but now even more with the support of major search engines. Hundreds of millions of documents already embed semi-structured data in the form of RDF, RDFa and Microformats and it is easy to predict that more will join soon. Whatever the current size of the Web of Data is today, the trend is clear and so is the requirement for handling semi-structured data with a scalability in the same class of traditional search engines.

However, the mass publishing of data on the Web is unexploitable by semantic clients and applications if supporting tools are not made available for data discovery. Taking the e-commerce example, how can a client find products matching a

⋆ The author contributed to this work while he was a master student in DERI

certain description pattern over thousands of e-commerce semantic data sources ? By entering keyword queries into a standard web search system, the results are likely to be irrelevant since the system will return pages mentioning the keywords and not the matching products themselves. Current search systems are inadequate for this task since they have been developed for a totally different model, i.e., a Web of Documents. The shift from documents to data entities poses new challenges for web search systems.

In this paper, we present the Semantic Information Retrieval Engine, SIREn, a system based on Information Retrieval (IR) techniques and designed to search "entities" specifically according to the requirements of the Web of Data. We advocate the use of a node indexing scheme for indexing semi-structured data, a technique coming from the XML Information Retrieval world. We analyse and compare the theoretical performances and other criteria of SIREn against three other indexing techniques for entity retrieval. We show that the node indexing scheme offers a good compromise between query expressiveness, query processing time and update complexity and scales well with very large datasets. The resulting system inherits from many characteristics of IR systems such as web like scalability, incremental updates and top-k queries among others.

1.1 Web of Data: Requirements for SIREn

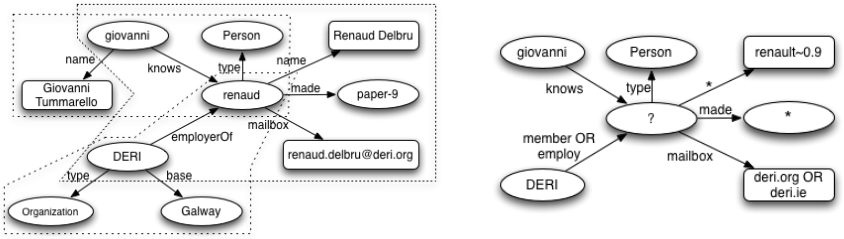
Developed within the Sindice project [1], SIREn is designed to be comparable in term of scalability to current web search engines so to be able to encompass, given sufficient hardware, the entire "Web of Data". The requirements are therefore:

1. Support for the multiple formats which are used on the Web of Data;
2. Support for searching an entity description given its characteristics (entity centric search);
3. Support for context (provenance) of information: entity descriptions are given in the context of a website or a dataset;
4. Support for semi-structural queries with full-text search, top-k query results, scalability over shard clusters of commodity machines, efficient caching strategy and incremental index maintenance.

With respect to point 1, the two formats which enable the annotation of entities on web pages are Microformats and RDF (RDFa is treated equally to RDF). At knowledge representation level, the main difference between Microformats and RDF is that the former can be seen as a frame model while the latter has a graph based data model. While these are major conceptual differences, it is easy to see that the RDF model can be used effectively to map Microformats[4]. Under these conditions, we have developed SIREn to cover the RDF model knowing that this would cover Microformats and likely other forms of web metadata.

With respect to point 2 and 3, the main use case for which SIREn is developed is entity search: given a description of an entity, i.e. a star-shaped query such as the one in Fig. 1(b), locate the most relevant entities and datasets. The Fig. 1(a) shows an RDF graph and how it can be split into three entities *renaud*, *giovanni* and *DERI*. Each entity description forms a sub-graph containing the incoming and outgoing relations of the entity node which is indexed by the system.

[4] Any23: `http://code.google.com/p/any23/`



(a) Visual representation of an RDF graph. The RDF graph is divided (dashed lines) into three entities identified by the nodes *renaud*, *giovanni* and *DERI*

(b) Star-shaped query matching the entity *renaud* where *?* is the bound variable and $\star$ a wildcard

Fig. 1. In these graphs, oval nodes represent resources and rectangular ones represent literals. For space consideration, URIs have been replaced by their local names.

Finally, we will see in Sect. 4 that SIREn leverages well known IR techniques to address the point 4.

1.2 Approaches for Entity Retrieval

Two main approaches have been taken for entity retrieval, based either on Database techniques or on Information Retrieval techniques.

Database and Retrieval of RDF Data Typically, entities described in RDF are handled using systems referred to as "triplestores" or "quadstores" which usually employ techniques coming from the Database world. Some of these are built on top of existing Relational Database such as Virtuoso[5] or on top of column stores [2] while others are purposely built to handle RDF [3–5].

These triplestores are built to manage large amounts of RDF triples[6] or quads[7] and they do so by employing multiples indices (generally B+-Trees) for covering all kind of access patterns of the form `(s,p,o,c)`. As for Database Management Systems, the main goal of these systems is answering complex queries, e.g. those posed using the SPARQL query language[8]. The task is a superset of entity retrieval as we defined it, and comes at the cost of maintaining complex data structures. Also they usually do not support natively top-k and full-text queries.

Information Retrieval for Semi-Structured Data In the past decades, many models [6] for textual database have been designed to support queries integrating content (words, phrases, etc.) and structure (for example, the table of contents). With the increasing number of XML documents published on the Web, new structured retrieval models and query languages such as XPath/XQuery [7] have been designed. Various indexing techniques [8] have been developed to optimise the processing of the XPath query language. Amongst them, the node indexing scheme

[5] Virtuoso: `http://virtuoso.openlinksw.com/`

[6] specifically a triple is a statement s, p, o consisting of a subject, a predicate, and an object and asserts that a subject has a property with some value.

[7] specifically a quad is a statement with a fourth element c called "context" for naming the RDF graph, generally to keep the provenance of the RDF data.

[8] SPARQL: `http://www.w3.org/TR/rdf-sparql-query/`

[9, 10] relies on node labelling schemes [11] to encode and query the tree structure of an XML document using either a database or an inverted index.

Other communities [12–14] have investigated IR techniques for searching semi-structured data. [14] investigates the use of an inverted index over string sequences to support search over loosely structured datasets. Semplore [12] extends inverted index to encode RDF graph approximation and supports tree-shaped queries over RDF graphs. The first system relies on a *field-based indexing* scheme to encode attribute-value relations into the index dictionary. The second uses multiple inverted indexes to encode various structural aspects of RDF. An analysis of their limitations and advantages are discussed in Sect.5.

In the context of the Semantic Web, we are aware of one work [15] that explores the use of node labelling schemes for indexing and querying voluminous subsumption hierarchies. In comparison to SIREn, this work has focused on label querying using standard relational DBMS for subsumption check in large RDF taxonomies and can not be directly applied for the entity retrieval problem.

1.3 Our Contribution

Our goal is to develop an entity retrieval system that supports the previously defined requirements. In this paper, we provide the following contributions towards this goal:

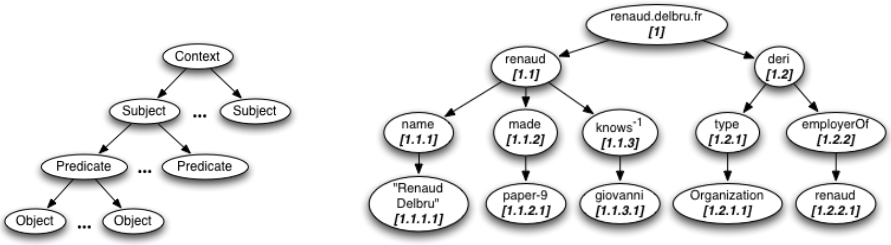
- We introduce a system based on a node indexing scheme and Information Retrieval techniques for searching semi-structured representation of entities;
- We describe how a node indexing scheme can capture a semi-structured representation of an entity as well as its provenance and how it can be implemented in a inverted index;
- We compare the node indexing scheme to three other schemes. We analyse their theoretical performances, present experimental results and show that the node indexing scheme scales well with a large number of triples.

The paper is organized as follows: we first present the node-labelled data model in Sect. 2 and the associated query model in Sect. 3. We describe in Sect. 4 how to extend inverted lists as well as update and query processing algorithms to support the node index model. An analysis of the differences and theoretical performance between SIREn and other entity retrieval systems is given in Sect. 5. In Sect. 6, experimental results are shown using large real world data collections and against other well known RDF management systems.

2 Node-Labelled Tree Model for RDF

A node-labelled tree model enables to efficiently establish relationships between nodes. The two main types of relations are *Parent-Child* and *Ancestor-Descendant* which are also core operations in XML query languages such as XPath. To support these relations, the requirement is to assign unique identifiers (node labels) that encode relationships between the nodes. Several node labelling schemes have been developed [11] but in the rest of the paper we will use a simple prefix scheme, the *Dewey Order* encoding [16]. With Dewey Order, each node is assigned a vector that represents the path from the tree's root to the node and each component of the path represents the local order of an ancestor node.

Using this labelling scheme, structural relationships between elements can be determined efficiently. An element u is an ancestor of an element v if label(u) is



(a) Conceptual representation of the node-labelled tree model

(b) Node-labelled tree model of the example dataset using Dewey's encoding

Fig. 2. The node-labelled tree model

a prefix of label(v). Fig. 2(b) presents a data tree where nodes have been labelled using Dewey's encoding. Given the label $\langle 1.2.1.1 \rangle$ for the term `Organisation`, we can find that its parent is the predicate `rdf:type`, labelled with $\langle 1.2.1 \rangle$.

SIREn adopts a node-labelled tree model to capture datasets, entities and their RDF descriptions. The tree model is pictured in Fig. 2(a). The model has four different kind of nodes: context (dataset), subject (entity), predicate and object. Each node can refer to one or more terms. In our case, a term is not necessarily a word (from a RDF Literal), but can be an URI or a local blank node identifier.

The node-labelled model covers the quad relations CSPO (outgoing relations) and COPS (incoming relations). Incoming relations are symbolised by a predicate node with a $^{-1}$ tag in Fig. 2(b). This model is not limited to quad relations, and could in theory be used to encode longer paths such as 2-hop relations but this is beyond the scope of this paper.

3 Query Model

Since RDF is semi-structured, we aim to support three types of queries: 1. full-text search (keyword based) when the data structure is unknown, 2. semi-structural queries (complex queries specified in a star-shaped structure) when the data schema is known, 3. or a combination of the two (where full-text search can be used on any part of the star-shaped query) when the data structure is partially known. In this section, we present a set of query operators over the content and structure of the node-labelled tree which covers the three types of queries. We will present the operators of SIREn and whenever possible compare them with their SPARQL equivalents (in Listing 1.1).

Content operators The content query operators are the only ones that access the content of a node, and are orthogonal to the structure operators. They include extended boolean operations such as boolean operators (intersection, union, difference), proximity operators (phrase, near, etc.) and fuzzy or wildcard operators.

These operations allow to express complex keyword queries for each node of the tree. Interestingly, it is possible to apply these operators not only on literals, but also on URIs if they are normalized (i.e., tokenized). For example one could just use the local name, e.g. `name`, to match `foaf:name` ignoring the namespace.

Structure operators In the following, we define a set of operations over the node-labelled tree. Thanks to these operations, we are able to search content to limited nodes, to query node relationships and to retrieve paths of nodes matching a

given pattern. Combination of nodes are possible using set operators, enabling the computation of entities and datasets matching a given star-shaped query.

Ancestor-Descendant: A//D A node A is the ancestor of a node D if it exists a path between A and D. For example, the SPARQL query in Listing 1.1, line 1, can be interpreted as an Ancestor-Descendant operator, line 2, and will return the path $\langle 1.2.2.1 \rangle$.

Parent-Child: P/C A node P is the parent of a node C if P is an ancestor of C and C is exactly one level above P. For example, the SPARQL query in Listing 1.1, line 3, can be translated into a Parent-Child operator, line 4, and will return the path $\langle 1.1.1.1 \rangle$.

Set manipulation operators These operators allow to manipulate nodes (context, subject, predicate and object) as sets, implementing union ($\cup$), difference ($\setminus$) and intersection ($\cap$). For example in Listing 1.1, the SPARQL query, line 5, can be interpreted as two PC and one intersection operators, line 6.

Compared to their XML equivalent, The AD and PC operators take in consideration the level of the nodes at query processing time in order to avoid false-positive results. For example, in Listing 1.1, line 2, the keywords `deri` and `renaud` are restricted to match subject nodes and object nodes respectively. Also, operators can be nested to express longer path as shown in Listing 1.1, line 7 and 9. However, the later is possible only if deeper trees have been indexed, i.e. 2-hop relations of an entity.

Listing 1.1. SPARQL queries and their SIREn interpretation

```
1  SELECT DISTINCT ?g WHERE { GRAPH ?g { <deri> ?p <renaud> }}
2  deri // renaud
3  SELECT DISTINCT ?g ?s WHERE { GRAPH ?g { ?s <name> "Renaud Delbru" }}
4  name / "Renaud Delbru"
5  SELECT DISTINCT ?g ?o WHERE { GRAPH ?g { <giovanni> <knows> ?o . <deri> <employerOf> ?o . }}
6  knows^-1 / giovanni AND employerOf^-1 / deri
7  SELECT DISTINCT ?s WHERE { GRAPH <renaud.delbru.fr> { ?s <knows> <renaud> }}
8  renaud.delbru.fr // knows / renaud
9  SELECT DISTINCT ?g ?s WHERE { GRAPH ?g { ?s <employerOf> ?o . ?o <name> "renaud" . }}
10 employerOf // name / "renaud"
```

4 Implementing the Model

In this section, we present the data format of the inverted list and the related update and query processing algorithms. This inverted list, in addition of being able to capture quad information, has distinctive features such as efficient incremental updates of entities in an existing context and self-indexing over contexts and subjects for faster access.

4.1 Inverted Lists

We will now explain how the structural information associated with a term can be transposed into a postings list. The format of the inverted list is similar to the path-based model described in [17]. In this model, the inverted list stores a term occurrence with a path from the root node (context) to the node that contains the word. For example, for a term that appears in a predicate the associated path will be $\langle$*context, subject, predicate*$\rangle$ while the path for a term that appears in an object will be $\langle$*context, subject, predicate, object*$\rangle$.

The inverted index I is built as a collection of n inverted lists $I_{t_0}, I_{t_1}, \ldots, I_{t_n}$ where a list I_t contains a posting for each occurrence of the term t in the data collection. A posting list holds a sequence of term occurrences in the format shown below. The path and positional information are stored in a term-interleaved [18] manner where various parts of the posting list are stored separately. In addition to provide effective compression, it enables incremental updates of an existing context as explained in the next section.

Listing 1.2. Posting list format

```
Term          -> <cid, tef, EntityInfo*>^tcf
EntityInfo    -> <sid, freq, NodeInfo*>^tef
NodeInfo      -> <pid, pos> | <pid, oid, pos> | ...
```

In Listing 1.2, each *Term* is associated to a first information block, an ordered list of context identifiers *cid* of size *tcf* ("term context frequency", the number of contexts mentioning the term). Each context identifier is immediately followed by *tef* and *EntityInfo* which correspond respectively to the "term entity frequency" (the number of entity in the context mentioning the term) and the pointer to the entity information block. The *EntityInfo* block is an ordered list of subject identifiers *sid* of length *tef*. Each subject identifier is immediately followed by the term frequency *freq* in this entity and a pointer *NodeInfo* to the information block containing remaining path and positional information for each term occurrence. The remaining path information of a term is defined by the predicate identifier *pid* and optionally by the object identifier *oid* if and only if the term belongs to an object. The position offset *pos* of the term within a node is also stored in order to enable phrase and proximity queries. Information is ordered first by predicate identifier then by object identifier and finally by position.

To produce compact posting lists, integers are stored as a difference, or delta representation [17], using variable-length byte encoding. The key idea of the delta compression is to store the difference between consecutive values instead of the values themselves. However, more advanced compression techniques [19] could be implemented instead.

4.2 Incremental Update of the Inverted Lists

The proposed model supports incremental updates of datasets and entities as it is performed for documents in traditional inverted indexes [20]. Adding a dataset or entity corresponds to adding a set of statements to the inverted index. The insertion of one quad (s,p,o,c) is performed by 1. accessing the postings list of the term p and o, and 2. appending to each postings list a new entry that contains the context identifier, the subject identifier as well as the related structural and positional information of the term. The interleaved structure of the posting list enables to add a new entity to an existing context. In that case, the information block containing the list of contexts is accessed in order to increment *tef* and retrieve the pointer of the related *EntityInfo* block. Then, a new entry is appended to the *EntityInfo* and *NodeInfo* blocks.

The complexity of insertion of one quad is $O(log(n) + 1)$, where $O(log(n))$ is the cost of searching a term in a dictionary of n terms and $O(1)$ is the cost of appending a posting to the list. When updates are performed by batches, the update time is linear with the number of postings to append.

Compared to triple stores, we do not support the deletion on a statement granularity, but we support the deletion of a context or subject, i.e. a set of statements. When a context or subject is removed, their identifier is inserted into a *deletion table*. During query processing, each posting entries is checked against the deletion table in $O(1)$ to ensure that it has not been deleted. The deletion table is integrated back to the inverted index only when a certain amount of deletion is sufficient to amortize the cost of such maintenance operation.

4.3 Query Processing

The evaluation of a query works in a bottom-up fashion. We first perform matching on the content (terms) of a node, then structural information is used during postings list intersection for filtering the result candidates that do not belong to the same node. The methodology for intersecting two postings lists is described by the following merge algorithm:

1. The postings list of each term is retrieved.
2. We then walk through the postings lists simultaneously.
3. At each step, we first compare the context and subject identifiers, then the predicate identifier and finally the object identifier. If they are the same, we put the pair `cid, sid` in the result list and advance the pointers to their next position in each postings list.

The worst-case complexity of a query evaluation is in time linear to the total number of posting entries [21]. In the average case, the complexity of an intersection is reduced to sub-linear time with an internal index (or skip lists [22]) over the context and subject identifiers to skip over unnecessary records.

Each query operator delivers output in sorted order. Multiple operators can be nested without losing the sorted order of the output, therefore enjoying the concept of interesting orderings [23] enabling the use of the effective merge-join without intermediate result sorting.

In addition, it is possible to apply existing scoring schemes such as TF-IDF or BM25F to compute top-k results at query time based on the keywords and structure of the matching sub-graphs. During the concurrent postings traversal we compute the score of one dataset-entity at a time, similarly to the document-at-a-time scoring in text database. However, more advanced top-k processing algorithms [20] could be employed instead.

5 Comparison among Entity Retrieval Systems

In this section, we evaluate four entity retrieval systems: SIREn based on a node-labelled index, field-based indexes [14], RDF databases [5] based on quad tables and Semplore [12]. These techniques are representative of the current approaches for entity retrieval.

Field-based indexing schemes are generally used in standard document retrieval systems (such as Apache Lucene) to support basic semi-structured information like document's field (e.g., the title). A field index constructs index terms by concatenating the field name (i.e., predicate URI) with the terms from the content of this field. For example, in the graph depicted in Fig. 1(a), the index terms for the entity "giovanni" and its predicate *name* will be represented as *name:giovanni* and *name:tummarello*.

Semplore is an Information Retrieval engine for querying Semantic Web data which supports hybrid queries, i.e. a subset of SPARQL mixed with full text search. Semplore is also built on inverted lists and relies on three inverted indexes: 1. an ontology index that stores the ontology graph (concepts and properties), 2. an individual path index that contains information for evaluating path queries, and 3. an individual content index that contains the content of the textual properties.

In the following, we assume that term dictionaries as well as quad tables are implemented with a b+-tree. The comparison is performed according to the following criteria: *Precision*, *Processing Complexity*, *Update Complexity* and *Query Expressiveness*. *Precision* evaluates if the system returns any false answers in the query result set. *Processing Complexity* evaluates the theoretical complexity for processing a query (lookups, joins, etc.). *Update Complexity* evaluates the theoretical complexity of maintenance operations. *Query Expressiveness* indicates the type of queries supported. Table 1 summarises the comparison.

Criteria	***Node Index***	***Field Index***	***Quad Table***	***Semplore***
Precision (false positive)	No	Yes	No	Yes
Dictionary Lookup	$O(log(n))$	$O(log(n*m))$	$O(log(n))$	$O(log(n))$
Quad Lookup	$O(log(n))$	$O(log(n*m))$	$O(log(n)+log(k))$	$O(log(n))$
Join in Quad Lookup	Yes	No	No	No
Star Query Evaluation	Sub-Linear	Sub-Linear	$O(n)$	$O(n*log(n))$
Update Cost	$O(log(n))$	$O(log(n*m))$	$O(log(n)+log(k))$	$O(log(n)+log(l))$
Multiple Indices	No	No	Yes	Yes
Query Expressiveness	Star	Star	Graph	Tree
Full-Text	Yes	Yes (on literals)	No	Yes (on literals)
Multi-Valued Support	Yes	No	Yes	No

Table 1. Summary of comparison among the four entity retrieval systems

Precision The field indexing scheme encodes the relation between predicates and terms in the index dictionary, but loses an important structural information: the distinction between literal objects. As a consequence, if the predicate is multi-valued, the field index may return false-positive results. Semplore suffers from a similar problem: it aggregates all the literal objects of an entity, disregarding the predicate, into a single bag of words. On the contrary, the node index and the quad table are able to distinguish distinct objects and do not produce wrong answers.

Processing Complexity Since the field-based index encodes relations between predicate and terms in the dictionary, its dictionary may quickly become large when dealing with heterogeneous data. A dictionary lookup has a complexity of $O(log(n*m))$ where n is the number of terms and m the number of predicates. This overhead can have a significant impact on the query processing time. In contrast, the other systems stay with a term dictionary of size n.

To lookup a quad or triple pattern, the complexity of the node and field index is equal to the complexity of looking up a term in the dictionary. In contrast, the RDF databases should perform in addition a lookup on the quad table. The complexity is $O(log(n)+log(k))$ with $log(n)$ the complexity to lookup a term in the dictionary and $log(k)$, k being the number of quads in the database, the complexity to lookup a quad in a quad table. In general, it is expected to have considerably more quads than terms, which can have a substantial impact on the query processing time for very large data collection. However, for quad patterns containing two or more terms, for example (?c,?s,p,o), the node index has to perform a merge-join between the posting lists of the two terms in order to check

their relationships, but such joins can be performed on average in sub-linear time. On the contrary, the other indexes do not have to perform such joins. But, in the context of Semplore, access patterns where the predicate is not specified cause a full index scan.

For evaluating a star-shaped query (joining multiples quad patterns), each index has to perform a merge-join between the records of each quad patterns. Such join is linear with the number of records in the case of the quad table, and sub-linear in the case of the node and field index if they use skip-lists. In contrast, Semplore has often to resort to expensive sort before merge-join operations.

Update Complexity In terms of complexity of maintenance, in a b+-tree system the cost of insertion of one quad represents the cost of search of the leaf node (i.e., $O(log(n)+log(k))$), the cost of adding a leaf node if there is no available leaf node and the cost of rebalancing (overhead to keep the tree balanced). These operations become problematic with large indices and requires advanced optimizations [24] that in return cause degradations in query performance. In contrast, the cost of insertion for a node and field index is equal to the cost of a dictionary lookup as discussed in Sect. 4.2, which is $O(log(n))$ and $O(log(n * m))$ for the node index and the field index respectively. Furthermore, quad tables are specific to access patterns, hence multiple b+-tree indexes have to be updated. Concerning the size of the indexes, all of them are linear with the data.

Concerning Semplore, the original system could not perform updates or deletions of triples without a full re-indexing. The authors have recently [12] proposed an extension for incremental maintenance operations based on the *landmark* [25] technique but the update complexity remains substantial. The update cost is $O(log(n) + log(l))$ with l the number of landmarks in the posting list. The fact that Semplore uses multiple indexes and landmarks considerably increase the update complexity. For example, index size and creation time reported in [12] are higher than for RDF-3X [5].

Query Expressiveness In term of query expressiveness, RDF databases have been designed to answer complex graph-shaped queries which are a superset of the queries supported by the other systems. On the other hand, the other systems are especially designed to support natively full-text search which is not the case for quad table indexes. Node indexes provide more flexibility in term of full-text search since it enables keyword search on every parts of a quad. In addition, node indexes support set operations on nodes that give the ability to express set-valued queries on both URI and literal multi-valued properties.

Semplore supports relational tree-shaped queries but loses structural information since the relation between a resource and a literal is not indexed. Hence, it is not possible to restrict full-text search of a literal using a predicate, e.g. asking `(?s, <foaf:name>, "renaud")`.

6 Experimental Results

In this section, we compare the performance of SIREn against RDF databases (using quad tables over b+-tree indexes) based on some of the above criteria. We assess the space requirement, the index creation time and the query processing performance. The aim is to show the benefits of using a system like SIREn for web entity retrieval compared to common approaches based on RDF databases. While RDF databases are very efficient to answer complex queries, we show that for the

(a) Index size in MB per dataset and system

	SIREn	Sesame	RDF-3X
Barton	789	3400	3076
Real-World	296	799	1138

(b) Indexing time in minutes per dataset and system

	SIREn10	SIREn100	Sesame	RDF-3X
Barton	3	1.5	266	11
Real-World	1	0.5	47	3.6

Table 2. Report on index size and indexing time

simpler task of entity retrieval, carefully designed systems can provide substantial benefits in term of scalability while sustaining fast query time.

The experimental setup is as follows. SIREn is implemented on top of Apache Lucene 2.4. The first RDF database is Sesame 2.0 with native backend (based on b+-tree), an open-source system which is commonly used as baseline for comparing quad store performances (e.g., in [5]). The second system is the state-of-the-art triple store RDF-3X [5]. We report that it is impossible for us to compare Semplore because at the time of the writing it is not being made available for this purpose. We also do not compare field-based index due to their query expressiveness limitations. In a previous publication [26], we reported experimental results showing the decrease of performance of field-based index compared to SIREn when the number of fields increases.

For the experiments, we use two datasets. The first one, called "Real-World" has been obtained by random sampling the content of the Sindice search engine. The real world dataset consists of 10M triples (approximately 2GB in size), and contains a balanced representation of triples coming from the Web of Data, e.g. RDF and Microformats data published online. The second dataset is the MIT Barton dataset that consists of 50M triples (approximately 6GB in size).

The machine that served for the experiment was equipped with 8GB ram, 2 quad core Intel processors running at 2.23 GHz, 7200 RPM SATA disk, Linux 2.6.24-19, Java 1.6.0.06 and GCC 4.2.4. All the following benchmarks are performed with cold-cache by flushing the kernel cache and by reloading the application after each query.

6.1 Index Size

The first experiment compares the index size of the three systems. The index size comprises the lexicon and the indices. Sesame is configured to create a single quad table (p,o,c,s). RDF-3X creates all the possible triple tables plus additional tables for query optimizations. SIREn creates a single inverted index.

The results are shown in Table 2(a). With respect to SIREn, Sesame exhibits at least a two-fold increase in index size on the real-world dataset and a four-fold increase on Barton. RDF-3X exhibits a four-fold increase in index size on the two datasets. With respect to the original dataset size, we observe that SIREn exhibits a index size ratio of 13-15%, whereas for Sesame and RDF-3X the ratio is approximately 50%. While the index size is linear with the size of the data collection for all the systems as discussed in Sect. 5, we can observe that the duplication of indices in RDF databases is causing a significant increase in index size compared to SIREn.

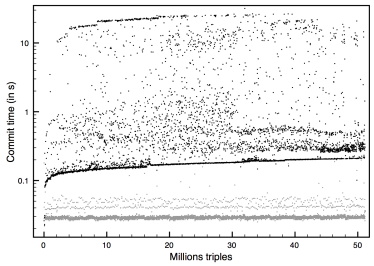
6.2 Insertion Time

In Table 2(b), we report the index creation time for the two datasets. For SIREn we report two cases: SIREn10 is the time to construct the index by batch of 10000 triples while SIREn100 is the time by batch of 100000 triples. Concerning

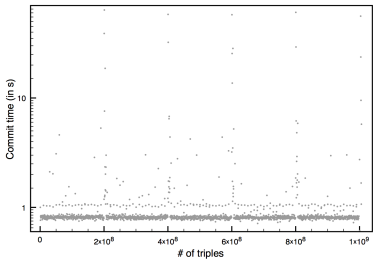
RDF-3X, it is important to notice that it does not support context, therefore it indexes triples and not quads, and that it does not support incremental indexing; RDF-3X needs the full dataset beforehand in order to construct the indexes in a single batch process, as opposed to Sesame which supports incremental updates. We can see from the results in Table 2(b) that SIREn is 50 to 100 times faster than Sesame and 3 to 6 times faster than RDF-3X.

In the next test, we plot the performance of SIREn and Sesame in an incremental update benchmark. The Fig. 3(a) shows the commit times for an incremental 10.000 triples batch on the two systems[9]. The graph is reported in logarithmic scale. While the absolute time is significant, the important result is the constant time exhibited by SIREn for incremental updates, as compared to Sesame performance which progressively decreases as the number of quads increases (as explained in Sect. 5).

In Fig. 3(b), the commit time of SIREn is plotted for a synthetic dataset constructed by replicating Barton 20 times so to reach 1 billion triples. The total index creation time is 31 minutes. We can notice that SIREn keeps a constant update time during the entire indexing. Outliers are due to periodic merges of the index segments. These results show that SIREn scales well with a large number of triples and provides significant improvement in terms of incremental update compared to other RDF databases.



(a) Plots showing the commit time every 10.000 triples during the index creation on Barton



(b) Plots showing the commit time every 500.000 triples during the index creation over one billion triples

Fig. 3. Dark dots are Sesame commit time records while gray dots are SIREn commit time records

6.3 Query Time Execution

For the query time execution benchmark, we created sets of queries with increasing complexity. The first set of queries (A*) consist of simple term lookups (URIs or literals). The second set of queries (B*) contains triple pattern lookups. The other sets consist of a combination of triple patterns using different set operators (intersection: C*, union: D*, exclusion: E). The queries are available at `http://siren.sindice.com`. For each query we average 50 query execution times without considering the final mapping between the result ids and their string values. The results are shown in Table 3(a) for Barton dataset and in Table 3(b) for Real-World dataset.

[9] We omit the commit times for the Real-World dataset since the results were similar to the Barton dataset

(a) Barton dataset

	A1	A2	B1	C1	C2	D1	D2	E
RDF-3X	16.12	0.12	1.38	1.16	0.38	0.23	0.14	X
SIREn	2.79	0.02	1.33	1.71	0.95	0.36	0.03	0.96

(b) Real-World dataset

	A1	A2	B1	B2	C1	C2	D1	E
RDF-3X	0.29	0.12	0.17	0.18	0.21	0.13	0.28	X
SIREn	0.23	0.03	0.04	0.05	0.09	0.08	0.16	0.53

(c) 10 Billion Triples dataset

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Time (s)	0.75	1.3	1.4	0.5	1.5	1.6	4
Hits	7552	9344	3.5M	57K	448	8.2M	20.7M

Table 3. Querying time in seconds

Since RDF-3X does not support native full-text search, we were unable to test queries involving this aspect. With respect to query E only SIREn was able to execute it since RDF-3X does not support the `bound` operator that is necessary to implement *exclusion* of triple patterns. With respect to Sesame, we decided not to include it in this test as during the experimentation phase we found that the results that we have obtained were consistently outperformed by RDF-3X.

The first observation is that on the Real-World dataset, SIREn performs significantly better, approximately 2 to 4 times, in 6 queries out of 7 while performing similarly in one, A1, a query which produces a very large amounts of results. In these queries and due to skewness of real-world data, SIREn are able to take advantage of its sub-linear merge-join by skipping unnecessary record comparisons.

On the Barton dataset, we notice however that for 3 queries out of 7 SIREn performs approximately 5 to 6 faster than RDF-3X, while resulting slower but comparable in 3 out of 7. In a particular query, C2, SIREn under-performs approximately 3 times. The explanation is that this query uses multiple triple access patterns that requires SIREn to perform more term lookups and merge-joins compared to RDF-3X and is therefore more expensive in term of disk I/O.

6.4 10 Billion Triples on a Single Machine

We evaluate SIREn scalability by indexing a dataset composed by 1 billion entities described in approximately 10 billion triples (one Terabyte of data). The dataset is derived from the billion triple challenge dataset[10]. To avoid hitting the limit of 2 billion entities due to the current implementation, we remove entities with only one or two triples and duplicate the remaining triples to reach 10 billion.

Since the dataset is different from the one in previous experiments, we use a different albeit comparable set of queries which is also provided at `http://siren.sindice.com`. The performance is given in the Table 3(c). Q1 to Q4 are property-object lookups using terms that are more or less frequent. In Q1 and Q2, we request for an infrequent property-object. The first query, while giving a result set of similar size, performs approximately two times better than Q2. The difference is that Q1 uses an infrequent predicate while Q2 a very frequent one, which in the latter case causes an overhead due to the merge-join. However, Q3 and Q4 use a very frequent property and, despite of the large increase of hits, the performance is similar or even better than Q2, which underlines that the complexity is linear with the length of the property posting list. Q5 performs a union between two infrequent property-object using a frequent property term. Again, we can observe the overhead caused by the merge-join between a frequent property and a term.

[10] Semantic Web Challenge: `http://challenge.semanticweb.org/`

However, Q6 and Q7 contain frequent properties and return a large number of results. The system scales linearly with the number of hits because the overhead of the join becomes less significant.

This scalability experiment shows that SIREn, even if there is a slight overhead when frequent properties are used in the query, scales well with a large number of triples and provides in all the cases reasonable query times, which makes it suitable for the web entity retrieval scenario.

7 Conclusion and Future Work

We presented SIREn, an entity retrieval system based on a node indexing scheme for searching the Web of Data. SIREn is designed for indexing and querying very large semi-structured datasets and offers constant time incremental updates and efficient entity lookup using semi-structural queries with full-text search capabilities. With respect to Database and Information Retrieval systems, SIREn positions itself somewhere in the middle as it allows semi-structural queries while retaining many desirable Information Retrieval features: single inverted index, effective caching, top-k queries and efficient index distribution over shards.

We demonstrated that a node indexing scheme provides a good compromise between query expressiveness, query processing time and update complexity. While such approach has an overhead during quad lookups due to additional joins, it provides fast enough answer time and scales well to a very large number of triples. Future works will concentrate on how to reduce the overhead of merge-joins in quad lookups, and on how to extend traditional weighting schemes to take into account RDF structural elements.

SIREn has been implemented and is in production at the core of the Sindice semantic search engine. At the time of the writing, SIREn serves over 60 million harvested web pages containing RDF or Microformats and answers several tens of thousands queries per day on a single machine.

8 Acknowledgments

This material is based upon works supported by the European FP7 project *Okkam - Enabling a Web of Entities* (contract no. ICT-215032), and by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

References

1. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: A document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies **3**(1) (2008)
2. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
3. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Proceedings of the 6th International Semantic Web Conference. Volume 4825 of Lecture Notes in Computer Science., Springer Verlag (November 2007) 211–224
4. Weiss, C., Karras, P., Bernstein, A.: Hexastore - sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment **1**(1) (2008) 1008–1019
5. Neumann, T., Weikum, G.: RDF-3X - a RISC-style Engine for RDF. Proceedings of the VLDB Endowment **1**(1) (2008) 647–659

6. Baeza-Yates, R., Navarro, G.: Integrating contents and structure in text retrieval. SIGMOD Rec. **25**(1) (1996) 67–79
7. Walsh, N., Fernández, M., Malhotra, A., Nagy, M., Marsh, J.: XQuery 1.0 and XPath 2.0 data model (XDM). W3C recommendation, W3C (January 2007)
8. Gang, G., Chirkova, R.: Efficiently Querying Large XML Data Repositories: A Survey. IEEE Transactions on Knowledge and Data Engineering **19**(10) (2007) 1381–1403
9. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In: Proceedings of the 27th International Conference on Very Large Data Bases. (2001) 361–370
10. Haixun, W., Hao, H., Jun, Y., Yu, P., Yu, J.: Dual Labeling: Answering Graph Reachability Queries in Constant Time. In: Proceedings of the 22nd International Conference on Data Engineering, IEEE (2006) 75–75
11. Su-Cheng, H., Chien-Sing, L.: Node Labeling Schemes in XML Query Optimization: A Survey and Trends. IETE Technical Review **26**(2) (2009) 88
12. Wang, H., Liu, Q., Penin, T., Fu, L., Zhang, L., Tran, T., Yu, Y., Pan, Y.: Semplore: A scalable IR approach to search the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web **7**(3) (2009) 177–188
13. Bast, H., Chitea, A., Suchanek, F., Weber, I.: ESTER: efficient search on text, entities, and relations. In: Proceedings of the 30th annual international ACM SIGIR conference, New York, NY, USA, ACM (2007) 671–678
14. Dong, X., Halevy, A.: Indexing dataspaces. Proceedings of the 2007 ACM SIGMOD international conference on Management of data (2007) 43
15. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On labeling schemes for the semantic web. In: Proceedings of the 12th international conference on World Wide Web. (2003) 544
16. Beyer, K., Viglas, S.D., Tatarinov, I., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered xml using a relational database system. In: Proceedings of the 2002 ACM SIGMOD international conference. (2002) 204–215
17. Sacks-davis, R., Dao, T., Thom, J.A., Zobel, J.: Indexing documents for queries on structure, content and attributes. In: Proceedings of International Symposium on Digital Media Information Base, World Scientific (November 1997) 236–245
18. Anh, V.N., Moffat, A.: Structured index organizations for high-throughput text querying. In: Proceedings of the 13th International Conference on String Processing and Information Retrieval. (2006) 304–315
19. Witten, I.H., Moffat, A., Bell, T.C.: Managing gigabytes (2nd ed.): compressing and indexing documents and images. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
20. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computer Surveys **38**(2) (2006) 6
21. Manning, C.D., Raghavan, P., Schtze, H.: Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA (2008)
22. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst. **14**(4) (1996) 349–379
23. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys **25**(2) (1993) 73
24. Graefe, G.: B-tree indexes for high update rates. ACM SIGMOD Record **35**(1) (2006) 39
25. Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Agarwal, R.: Dynamic maintenance of web indexes using landmarks. In: Proceedings of the 12th international conference on World Wide Web. (2003) 102
26. Delbru, R., Toupikov, N., Catasta, M., Fuller, R., Tummarello, G.: SIREn: Efficient Search on Semi- Structured Documents. In: Lucene in Action, Second Edition. Manning Publications Co. (2009)