Searching Web Data: an Entity Retrieval and High-Performance Indexing Model

Renaud Delbru[a], Stephane Campinas[a], Giovanni Tummarello[a,b]

[a]*Digital Enterprise Research Institute,*
National University of Ireland, Galway
Galway, Ireland
[b]*Fondazione Bruno Kessler*
Trento, Italy

Abstract

More and more (semi) structured information is becoming available on the Web in the form of documents embedding metadata (e.g., RDF, RDFa, Microformats and others). There are already hundreds of millions of such documents accessible and their number is growing rapidly. This calls for large scale systems providing effective means of searching and retrieving this semi-structured information with the ultimate goal of making it exploitable by humans and machines alike.

This article examines the shift from the traditional web document model to a web data object (entity) model and studies the challenges faced in implementing a scalable and high performance system for searching semi-structured data objects over a large heterogeneous and decentralised infrastructure. Towards this goal, we define an entity retrieval model, develop novel methodologies for supporting this model and show how to achieve a high-performance entity retrieval system. We introduce an indexing methodology for semi-structured data which offers a good compromise between query expressiveness, query processing and index maintenance compared to other approaches. We address high-performance by optimisation of the index data structure using appropriate compression techniques. Finally, we demonstrate that the resulting system can index billions of data objects and provides keyword-based as well as more advanced search interfaces for retrieving relevant data objects in sub-second time.

This work has been part of the Sindice search engine project at the Digital Enterprise Research Institute (DERI), NUI Galway. The Sindice system currently maintains more than 200 million pages downloaded from the Web and is being used actively by many researchers within and outside of DERI.

Keywords: Sindice, Information Retrieval, Entity Retrieval, Indexing Scheme, Compression, Semantic Web, Web Data, Semi-Structured Data, RDF, Inverted Index, Search Engine, Entity Search

1. Introduction

More and more structured and semi-structured data sources are becoming available. With the current availability of data publishing standards and tools, publishing semi-structured data on the Web, which from here on we will simply call *Web Data*, is becoming a mass activity. Indeed, nowadays it is not limited to a few trained specialists any more. Instead, it is open to industries, governments and individuals. For example, the Linked Open Data[1] community has made available hundreds of data sources, driven by the idea of open access to data. There are also many prominent fields in which examples of semi-structured data publishing efforts exist: e-government, e.g., the UK government which publicly shares open government data; editorial world, e.g., the New York Times or Reuters which publish rich metadata about their news articles; e-commerce, e.g., BestBuy which publishes product descriptions in a machine-readable format; social networking, e.g., Facebook's Open Graph Protocol which enables any web page to become a rich object in a social graph; etc.

However, the usefulness of Web Data publishing is clearly dependent of the ease by which data can be discovered and consumed by others. Taking the e-commerce example, how can a user find products matching a certain description pattern over thousands

Email addresses: renaud.delbru@deri.org (Renaud Delbru), stephane.campinas@deri.org (Stephane Campinas), giovanni.tummarello@deri.org (Giovanni Tummarello)

[1] Linked Data: http://linkeddata.org/

of e-commerce data sources ? By entering simple keywords into a 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 objects* poses new challenges for web search systems. One of the main challenge is to develop efficient and effective methods for searching and retrieving data objects among decentralised data sources.

1.1. Information Retrieval for Web Documents

Information Retrieval systems for the Web, i.e., web search engines, are mainly devoted to finding relevant web documents in response to a user's query. Such a retrieval system is following a *document-centric* model, since the whole system is organised around the concept of document. The representation of a document which is commonly adopted is the *full-text* logical view [1] where a document is seen as a set or sequence of words.

With the scale and nature of the Web, Information Retrieval systems had to face new problems. Web search engines have to provide access to billions of documents. Indexing and query processing at this scale requires advanced techniques to provide good quality results in sub-second response time [2]. While Information Retrieval systems for Web Data have to face similar problems, they have to deal with an additional factor: the information and search requests are semi-structured instead of being unstructured. Moving from unstructured to semi-structured information raises new challenges but also provides new opportunities for developing more advanced search techniques.

1.2. Information Retrieval for Web Data

Compared to web documents, web data is providing some kind of structure. However, since each of the web data sources has its own defined schema, ranging from loosely to strictly defined, the data structure does not follow strict rules as in a database. Even in one data source, the schema might not be fixed and may change as the information grows. The information structure evolves over time, and new records can require new attributes. We therefore consider Web Data as being *semi-structured* [3].

The semi-structured information items that are found on the Web are very diverse. These items can describe a person, a product, a location, a document, etc. In fact, these items can describe any kind of *entities*. We consider these items, which from here on we will simply call *entity descriptions*, as the principal units of information to be searched and retrieved. Compared to a document, an entity description is a more complex data object which is composed of a set of attribute and value pairs and possibly a set of relations to other entities. We envision that the task of an Information Retrieval system for Web Data is to provide a list of relevant entity descriptions in response to some information request which can be semi-structured itself. This task is indeed a valuable one. A great fraction of the user's queries (more than half of them) are entity-centric [4], and the potential of web data for this task is immense compared to that of web documents. For these reasons, we organise our retrieval model around the concept of an *entity* as opposed to the concept of a *document* found in traditional web search engines, and design the query language so to be able to support the needs of entity search (e.g., restrictions on attributes). We say that the retrieval system is following an *entity-centric* model.

1.3. Challenges in Information Retrieval for Web Data: Contributions

In this article, we study the challenges in building a large scale and high performance Information Retrieval system for Web Data. By large scale, we mean a system that can be deployed on a very large number of machines and that grows gracefully as the data and user volume increases. By high-performance, we mean a system that handles a continuous flow of update requests and that answers many queries in sub-second time.

A first problem in building such a retrieval system is to cope with the shift from the traditional document-centric model to the entity-centric model. There is a need to reassess the retrieval model as well as the boolean search model which does not provide foundations for querying semi-structured data. Section 3 which introduces our entity retrieval model undertakes precisely that task.

Also, the retrieval system has to offer advanced search interfaces and provide an access to billions of entities in sub-second response times. New index structures as well as new index optimisations are needed to incorporate semi-structured information in the system while sustaining a fast query computation and an efficient index maintenance. As a first step towards that goal, we propose in Section 4 a node-based inverted index that supports our entity retrieval model. Then, in Section 5, we introduce a high-performance compression technique which has tremendous benefits with respect to the node-based indexing scheme. In Section 6,

we show that the combination of the compression technique with the node-indexing scheme results in a high-performance retrieval system that can index billions of entities while sustaining sub-second response time.

In Section 2, we review existing works from domains such as indexing and searching semi-structured data. Finally, Section 7 recalls the main findings of the research and discusses the tasks that remain.

2. Related Work

In this section, we first give a retrospective of retrieval systems for structured documents, before presenting an overview of the current approaches for indexing and querying RDF data. Finally, we review existing search models for semi-structured and structured data.

2.1. Retrieval of Structured Documents

The problem with a traditional document retrieval system is its inability to capture the structure of a document, since a document is seen as a "bag of words". In order to query the content as well as the structure of the document, many models have been developed to support queries integrating content (words, phrases, etc.) and structure (for example, the table of contents). Amongst them, we can cite the hybrid model, PAT expressions, overlapped lists and proximal nodes. We refer the reader to [5] for a comparison of these models. By mixing content and structure, more expressive queries become possible such as relations between content and structural elements or between structural elements themselves. For example, it becomes possible to restrict the search of a word within a chapter or a section, or to retrieve all the citations from a chapter.

With the increasing number of XML documents published on the Web, new retrieval systems for the XML data model have been investigated. Certain approaches [6, 7, 8] have investigated the use of keyword-based search for XML data. Given a keyword query, these systems retrieve document fragments and use a ranking mechanism to increase the search result quality. However, such systems are restricted to keyword search and do not support complex structural constraints. Therefore, other approaches [9, 10] have investigated the use of both keyword-based and structural constraints. In the meantime, various indexing techniques have been developed for optimising the processing of XPath queries, the standardised query language for XML [11]. The three major techniques are the *node index scheme* [12, 13, 14], the *graph index scheme* [15, 16, 17], and the *sequence index scheme* [18, 19, 20, 21, 22]. The node index scheme relies on node labelling techniques [23] to encode the tree structure of an XML document in a database or in an inverted index. Graph index schemes are based on secondary indexes that contain structural path summaries in order to avoid join operations during query processing. Sequence index schemes encode the structure into string sequences and use string matching techniques over these sequences to answer queries.

More recently, the Database community has investigated the problem of search capabilities over semi-structured data in Dataspaces [24]. [25] proposes a sequence index scheme to support search over loosely structured datasets using conditions on attributes and values. However, the sequence indexing scheme is inappropriate for large heterogeneous data collections as discussed in Section 4.4.

2.2. Retrieval of RDF Data

With the growth of RDF data published on the Semantic Web, applications demand scalable and efficient RDF data management systems. Different RDF storage strategies have been proposed based on RDBMS [26], using native index [27, 28, 29, 30] or lately using column-oriented DBMS using vertical partitioning over predicates [31].

RDF data management systems excel in storing large amounts of RDF data and are able of answering complex conjunctive SPARQL queries involving large joins. Typical SPARQL queries are graph patterns combining SQL-like logical conditions over RDF statements with regular-expression patterns. The result of a query is a set of subgraphs matching precisely the graph pattern defined in the query. This search and retrieval paradigm is well adapted for retrieving data from large RDF datasets with a globally known schema such as a social network graph or a product catalog database. However, such approaches are of very limited value when it comes to searching over highly heterogeneous data collections. The variance in the data structure and in the vocabularies makes it difficult to write precise SPARQL queries. In addition, none of the above approaches uses structural and content similarity between the query and the data graph for ordering results with respect to their estimated relevance with the query.

Other approaches [32, 33, 34, 35] carried over keyword-based search to Semantic Web and RDF data in order to provide ranked retrieval using content-based relevance estimation. In addition, such systems are more easy to scale due to the simplicity of their indexing scheme. However, such systems are restricted to keyword search and do not support structural constraints.

These systems do not consider the rich structure provided by RDF data. A first step towards a more powerful search interface combining imprecise keyword search with precise structural constraints has been investigated by K-Search [36] and Semplore [37]. Semplore [37] has extended inverted index to encode RDF graph approximations and to support keyword-based tree-shaped queries over RDF graphs. However, we will see in Section 4.4 that the increase of query capabilities comes at a cost, and the scalability of the system becomes limited.

2.3. Search Models for Semi-Structured Data

In addition to standardised query languages such as SPARQL, XQuery or XPath, a large number of search models for semi-structured data can be found in the literature. Some of them focus on searching structured databases [38, 39, 40, 41], XML documents [7, 6, 8] or graph-based data [42, 43, 44] using simple keyword search. Simple keyword-based search has the advantages of (1) being easy to use by users since it hides from the user any structural information of the underlying data collection, and (2) of being applicable on any scenarios. On the other hand, the keyword-based approach suffers from limited capability of expressing various degrees of structure when users have a partial knowledge about the data structure.

Other works [45, 46, 37, 25, 47] have extended simple keyword-based search with structured queries capabilities. In [25, 47], they propose a partial solution to the lack of expressiveness of the keyword-based approach by allowing search using conditions on attributes and values. In [45, 46, 37], they present more powerful query language by adopting a graph-based model. However, the increase of query expressiveness is tied with the processing complexity, and the graph-based models [45, 46, 37] are not applicable on a very large scale.

The search model introduced in Section 3 is similar to [25, 47], i.e., it is defined around the concept of attribute-value pairs. However, our model is more expressive since it differentiates between single and multi-valued attributes and it considers the provenance of the information.

3. An Entity Retrieval Model for Web Data*

In this section, we introduce an entity retrieval model for semi-structured information found in distributed and heterogeneous data sources. This model is used as a common framework to develop various methodologies for the entity retrieval system. For example in this article, we use it to design our indexing system. However, such a model has also been used for (1) designing a link analysis technique [49] for measuring the importance of an entity by exploiting the peculiar properties of links between datasets and entities, and (2) a distributed reasoning mechanism [48] which is tolerant to low data quality. We start by examining the Web Data scenario before introducing a data model that encompasses its core concepts. We finally discuss some requirements for searching semi-structured information and introduce our boolean search model and query algebra.

3.1. An Abstract Model for Web Data

For the purpose of this work, we use the following model for Web Data. We define Web Data as part of the Hypertext Transfer Protocol (HTTP) [50] accessible Web that returns semi-structured information using standard interchange formats and practices. The standard data interchange formats include HTML pages which embed RDFa or Microformats as well as RDF models using different syntaxes such as RDF/XML [51].

3.1.1. Practical Use Case

To support the Web Data scenario, we take as an example the case of a personal web dataset. A personal web dataset is a set of semi-structured information published on one personal web site. The web site is composed of multiple documents, either using HTML pages embedding RDFa or using plain RDF files. Each of these documents contains a set of statements describing one or more entities. By aggregating the semi-structured content of these documents altogether, we can recreate a dataset or RDF graph. Figure 1 depicts such a case. The content, or database, of the web site `http://renaud.delbru.fr/` is exposed through various accessible online documents, among them `http://renaud.delbru.fr/rdf/foaf` and `http://renaud.delbru.fr/publications.html`. The former document provides information about the owner of the personal web site and about its social relationships. The second document provides information about the publications authored by the owner of the web site. Each of them is providing complementary pieces of information which when aggregated provide a more complete view about the owner of the web site.

3.1.2. Model Abstraction

In the previous scenario, it is possible to abstract the following core concepts in semi-structured data web publishing: *dataset*, *entity* and *view*:

*This section is partially based on [48]

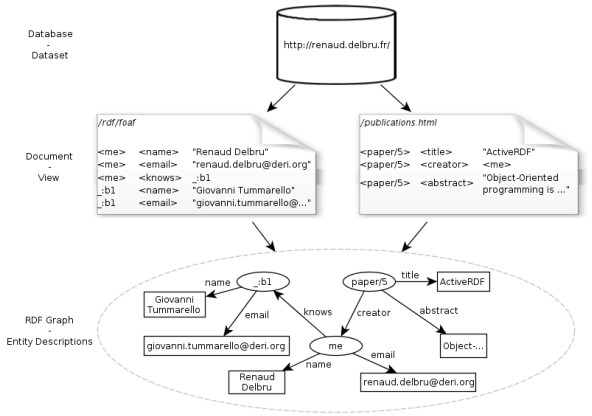


Figure 1: A sample of a personal web dataset. The dataset `http://renaud.delbru.fr/` makes available partial *view* of its content in the form of documents containing semi-structured data. The aggregation of all the views enables the reconstruction of the dataset in the form of a data graph.

A dataset is a collection of entity descriptions. One dataset is usually the content of a database which powers a web application exposing metadata, be this a dynamic web site with just partial metadata markups or a RDF database which exposes its content such as the Linked Open Data datasets. Datasets however can also come in the form of a single RDF document, e.g., an individual FOAF file posted on a person's homepage. A dataset is uniquely identified by a URI, e.g., `http://renaud.delbru.fr/` as depicted in Figure 1.

An entity description is a set of assertions about an entity and belongs to a dataset. The assertions provide information regarding the entity such as attributes, for instance, the firstname and surname for a person, or relationships with other entities, for instance, the family members for a person. An entity description has a unique identifier, e.g., a URI, with respect to a dataset.

A View represents a single accessible piece of information which provides a full or partial view over the dataset content. In the case of Linked Open Data, a typical view is the RDF model returned when one dereferences the URI of an entity. The Linked Open Data views are 1 to 1 mapping from the URI to the complete entity description[2]. This however is more the exception than the rule for other kind of Web Data publishing where most often only partial entity descriptions are provided in the views. For example in the case of Microformats or RDFa, views are pages that talk about different aspects of the entity, e.g., a page listing social contacts for a person, a separate page listing personal data as well as a page containing all the posts by a user.

[2]In Database terminology, this would be considered as a *Record*.

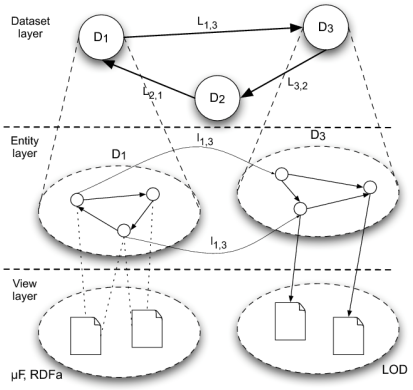


Figure 2: The three-layer model of Web Data

As shown in Figure 1, the union of all the views provided within a context, e.g., a web site, might enable an agent, e.g., a crawler, to reconstruct the entire dataset. There is no guarantee on this since it is impossible to know if we are in possession of every piece of information about an entity. The problem of retrieving the views and reconstructing a dataset is more related to data acquisition which is out of scope of this article.

These abstract concepts are used to define a three layer model, which is graphically represented in Figure 2. This model forms a common base for various methodologies developed towards the creation of an entity retrieval system. As mentioned before, the model is also used in other works [49, 48].

3.2. Formal Model

For the purpose of this work, we need a generic graph model that supports the previous scenario and covers the three layer model discussed previously. First, we define a labelled directed graph model that covers the various type of Web Data sources, i.e., Microformats, RDFa, RDF databases, etc. This graph model represents datasets, entities and their relationships. With respect

However, we think that the concept of *View* is more appropriate for the Web Data scenario given that entity descriptions are in fact the result of a join query returning a set of triples.

to the graph model, we define an *Entity Attribute-Value* model that will shape the indexing system described in Section 4.

3.2.1. Data Graph

Let V be a set of nodes and A a set of labelled edges. The set of nodes V is composed of two non-overlapping sets: a set of entity nodes V^E and a set of literal nodes V^L. Let $\mathcal{L}$ be a set of labels composed of a set of node labels $\mathcal{L}^V$ and a set of edge labels $\mathcal{L}^A$.

Web Data is defined as a graph G over $\mathcal{L}$, and is a tuple $G = \langle V, A, \lambda \rangle$ where $\lambda : V \rightarrow \mathcal{L}^V$ is a node labelling function. The set of labelled edges is defined as $A \subseteq \{(e, \alpha, v) | e \in V^E, \alpha \in \mathcal{L}^A, v \in V\}$. The components of an edge $a \in A$ will be denoted by $source(a)$, $label(a)$ and $target(a)$ respectively.

3.2.2. Dataset

A dataset is defined as a subgraph of the Web Data graph:

Definition 3.1 (Dataset)**.** A dataset D over a graph $G = \langle V, A, \lambda \rangle$ is a tuple $D = \langle V_D, A_D, \mathcal{L}_D^V, \lambda \rangle$ with $V_D \subseteq V$ and $A_D \subseteq A$.

We identify a subset $\mathcal{L}_D^V \subseteq \mathcal{L}^V$ of node labels to be *internal* to a dataset D, i.e., the set of entity identifiers and literals that originates from this dataset. For example, such a set might include the URIs and literals defined by the naming authority of the dataset [52].

Definition 3.2 (Internal Node)**.** A node $v \in V$ is said to be internal to a dataset D if $\lambda(v) \in \mathcal{L}_D^V$.

Analogously, we identify as *external* a node with a label that does not belong to $\mathcal{L}_D^V$.

Definition 3.3 (External Node)**.** A node $v \in V$ is said to be external to a dataset D if $\lambda(v) \notin \mathcal{L}_D^V$.

We assume that a literal node is always *internal* to a dataset D. A literal is always defined with respect to an entity and within the context of a dataset. Therefore, if two literal nodes have identical labels, we consider them as two different nodes in the graph G. This assumption does not have consequences in this paper. In fact, this can be seen as a denormalisation of the data graph which increases the number of nodes in order to simplify data processing.

Two datasets are not mutually exclusive and their entity nodes $V_D^E \subseteq V_D$ may overlap, i.e., $V_{D_1}^E \cap V_{D_2}^E \neq \emptyset$, since (1) two datasets can contain identical *external* entity nodes, and (2) the *internal* entity nodes in one dataset can be *external* entity nodes in another dataset.

3.2.3. Data Links

While the notion of links is mainly used in link analysis scenario [49], we describe it for the completeness of the model. In a dataset, we identify two types of edges: *intra-dataset* and *inter-dataset* edges. An *intra-dataset* edge is connecting two *internal* nodes, while an *inter-dataset* edge is connecting one *internal* node with one *external* node.

Definition 3.4 (Intra-Dataset Edge)**.** An edge $a \in A_D$ is said to be *intra-dataset* if $\lambda(source(a)) \in \mathcal{L}_D^V, \lambda(target(a)) \in \mathcal{L}_D^V$.

Definition 3.5 (Inter-Dataset Edge)**.** An edge $a \in A_D$ is said to be *inter-dataset* if $\lambda(source(a)) \in \mathcal{L}_D^V, \lambda(target(a)) \notin \mathcal{L}_D^V$ or if $\lambda(source(a)) \notin \mathcal{L}_D^V, \lambda(target(a)) \in \mathcal{L}_D^V$.

We group inter-dataset links into *linkset*. For example, in Figure 2 the inter-dataset links $l_{1,3}$ between D_1 and D_3 are aggregated to form the linkset $L_{1,3}$ on the dataset layer.

Definition 3.6 (Linkset)**.** Given two datasets D_i and D_j, we denote a linkset between D_i and D_j with $L_{\alpha,i,j} = \{a | label(a) = \alpha, source(a) \in D_i, target(a) \in D_j\}$ the set of edges having the same label α and connecting the dataset D_i to the dataset D_j.

3.2.4. Entity Attribute-Value Model

A dataset provides information about an entity including its relationships with other entities and its attributes. Consequently, a subgraph describing an entity can be extracted from a dataset.

The simplest form of description for an (internal or external) entity node e is a *star graph*, i.e., a subgraph of a dataset D with one central node e and one or more (inter-dataset or intra-dataset) edges. Figure 3 shows how the RDF graph from Figure 1 can be split into three entities *me*, *_:b1* and *paper/5*. Each entity description forms a sub-graph containing the incoming and outgoing edges of the entity node. More complex graphs might be extracted, but their discussion is out of scope of this paper.

An entity description is defined as a tuple $\langle e, A_e, V_e \rangle$ where $e \in V_D^E$ the entity node, $A_e \subseteq \{(e, \alpha, v) | \alpha \in \mathcal{L}_D^A, v \in V_e\}$ the set of labelled edges representing the attributes and $V_e \subseteq V_D$ the set of nodes representing values.

Conceptually, an entity is represented by an identifier, a set of attributes and a set of values for these attributes. Such a model is similar to the Entity Attribute Value model (EAV) [53] which is typically used for representing highly sparse data with many attributes. Under this

Figure 3: A visual representation of the RDF graph from Figure 1 divided into three entities identified by the nodes *me*, *_:b1* and *paper/5*.

me

Attribute	Value
name	Renaud Delbru
email	renaud.delbru@deri.org
knows	_:b1
creator $^{-1}$	paper/5

paper/5

Attribute	Value
title	ActiveRDF
abstract	Object-oriented programming is ...
creator	me

_:b1

Attribute	Value
name	Giovanni Tummarello
email	giovanni.tummarello@...
knows $^{-1}$	me

Figure 4: An example of EAV model derived from the three subgraphs from Figure 3.

model, we can depict the dataset example from Figure 1 as a set of EAV tables in Figure 4.

In its most simple form, an attribute is composed by one edge connecting two nodes, the central entity node e and a value node v. However, this is not always the case. More complex attributes such as multi-valued attributes are composed of more than one edge. We present and define in the following the possibles types of attributes we aim to support.

Attribute. An attribute represents an atomic characteristic of an entity, for example the name of a person or the title of a publication.

Definition 3.7 (Attribute)**.** Given an entity e, an attribute is defined as being an edge $(e, \alpha, v)|\alpha \in \mathcal{L}^A, v \in V_e$

Single-Valued Attribute. A single-valued attribute is an attribute that holds exactly one value. For example, the birthdate is a single-valued property since a person has only one birthdate.

Definition 3.8 (Single-Valued Attribute)**.** Given an entity e, a single-valued attribute is an attribute with exactly one value $(e, \alpha, v)|\exists! v\ target(\alpha) = v$.

Multi-Valued Attribute. An attribute is multi-valued when it has more than one value. For example, the email address of a person can be a multi-valued attribute since a person has possibly more than one email address.

Definition 3.9 (Multi-Valued Attribute)**.** Given an entity e, a multi-valued attribute is a set, with at least two members, of attributes having the same label $\alpha \in \mathcal{L}^A$ but with different value nodes.

Based on this Entity Attribute-Value data model, we introduce a boolean search model which enables the retrieval of entities. We define formally this search model with a query algebra.

3.3. Search Model

The search model presented here adopts a semi-structural logical view as opposed to the full-text logical view of traditional web search engines. We do not represent an entity by a bag of words, but we instead consider its set of attribute-value pairs. The search model is therefore restricted to search entities using a boolean combination of attribute-value pairs. We aim to support three types of queries:

full-text query: the typical keyword based queries, useful when the data structure is unknown;

structural query: complex queries specified in a star-shaped structure, useful when the data schema is known;

semi-structural query: a combination of the two where full-text search can be used on any part of the star-shaped query, useful when the data structure is partially known.

This gradual increase of query structure enables to accommodate various kind of information requests, from vague using full-text queries to precise using structural queries. The type of request depends on the awareness of the data structure by the user and on the user expertise.

The main use case for which this search model is developed is entity search: given a description pattern of an entity, i.e., a star-shaped queries such as the one in Figure 5, locate the most suitable entities and datasets. This means that, in terms of granularity, the search needs to move from a "document" centric point of view (as per normal web search) to a "dataset-entity" centric point of view.

3.3.1. Search Query Algebra

In this section, we introduce a formal model of our search query language. For clarity reasons, we adopt a model which is similar to the relational query algebra [54], where the inputs and outputs of each operator are "relations". All operators in the algebra accept one or two relations as arguments and return one relation as a result. We first describe the relations that are used in

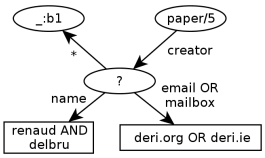


Figure 5: A star-shaped query matching the description graph of the entity *me* from Figure 3. *?* stands for the bound variable and ⋆ for a wildcard.

Dataset	*Entity*	*Attribute*	*Value*
delbru.fr	me	name	Renaud Delbru
delbru.fr	me	knows	_:b1
delbru.fr	paper/5	creator	me

(a) Entity Attribute Value relation R_1

Dataset	*Entity*	*Attribute*	*Value*
delbru.fr	paper/5	title	ActiveRDF
delbru.fr	paper/5	creator	me
delbru.fr	me	name	Renaud Delbru

(b) Entity Attribute Value relation R_2

Table 1: An example of two Entity Attribute Value relations, R_1 and R_2

the algebra before introducing the basic operators of the algebra.

Relations. For the purpose of the query algebra, we define an entity attribute value table as a relation $\langle dataset, entity, attribute, value\rangle$ where the fields

dataset holds the label of a dataset D;
entity holds the label of the entity node $e \in V_D^E$;
label holds the attribute label $\alpha \in \mathcal{L}^A$;
value holds the label of the value node.

Table 1a and Table 1b depict two relations that are derived from the EAV model of Figure 4. These two relations are used as inputs in the following examples.

In the next algebra formulas, we will use d to denote the fields *dataset*, e for *entity*, *at* for *attribute* and v for *value*.

Set Operations. Since relations are considered as sets, boolean operations such as union ($\cup$), intersection ($\cap$) and set-difference ($\setminus$) are applicable on relations.

Keyword Selection. The search unit is a keyword $k \in \mathcal{K}$ where $\mathcal{K}$ is the lexicon of the Web Data graph, i.e., the set of distinct words occurring in $\mathcal{L}$.

Let $W : \mathcal{L} \rightarrow \mathcal{K}$ a function that maps a label $l \in \mathcal{L}$ to a set of words $K \subset \mathcal{K}$. Given a keyword k, if $k \in W(l)$, it means that the word denoted by k appears at least one time in the label l. We say that k matches l.

The *Keyword Selection* operator σ is a unary operator. The selection operator allows to specify the relation instances to retain through a keyword selection condition.

Dataset	*Entity*	*Attribute*	*Value*
delbru.fr	me	name	Renaud Delbru

(a) $\sigma_{v:renaud}(R_1)$ or $\sigma_{v:\text{“}renaud\ delbru\text{”}}(R_1)$

Dataset	*Entity*
delbru.fr	me

(b) $\pi_{d,e}(\sigma_{v:renaud}(R_1))$

Table 2: An example showing the selection and projection operations

Definition 3.10 (Keyword Selection)**.** Given a keyword selection condition c and a relation R, the keyword selection operator $\sigma_c(R)$ is defined as a set of relation instances $\{r|r \in R\}$ for which the condition c is true.

The most basic form of a keyword selection condition is to test if a given keyword k occurs in one of the field f of a relation R, which is denoted by *f:k*. For example, one can test if the keyword k occurs in the dataset label of a relation instance r (denoted by *r.d*):

$$\sigma_{d:k}(R) : \{r|r \in R, k \in W(r.d)\}$$

or in the value node label of a relation instance r (denoted by *r.v*):

$$\sigma_{v:k}(R) : \{r|r \in R, k \in W(r.v)\}$$

The selection operator has the following properties. The proofs of these properties can be found in [54].

idempotent: multiple applications of the same selection operator have no additional effect beyond the first one as show in equation (1a).
commutative: the order of the selections has no effect on the result as show in equation (1b).
distributive: the selection is distributive over the set-difference, intersection and union operators as show in equation (1c), where $\gamma = \cap, \cup$ or $\setminus$.

$$\sigma_{f:k}(\sigma_{f:k}(R)) = \sigma_{f:k}(R) \tag{1a}$$
$$\sigma_{f:k_1}(\sigma_{f:k_2}(R)) = \sigma_{f:k_2}(\sigma_{f:k_1}(R)) \tag{1b}$$
$$\sigma_{f:k}(R \gamma S) = \sigma_{f:k}(R)\ \gamma\ \sigma_{f:k}(S) \tag{1c}$$

In general, the keyword selection condition is defined by a boolean combination of keywords using the logical operators $\wedge, \vee$ or $\neg$. A keyword selection using a boolean combination of keywords is identical to a boolean combination of keyword selections as shown in equations (2a), (2b) and (2c). Also, two nested selections are equivalent to an intersection of two selections as shown in equation (2d).

$$\sigma_{f:k_1}(R) \cap \sigma_{f:k_2}(R) = \sigma_{f:k_1 \wedge f:k_2}(R) \tag{2a}$$
$$\sigma_{f:k_1}(R) \cup \sigma_{f:k_2}(R) = \sigma_{f:k_1 \vee f:k_2}(R) \tag{2b}$$
$$\sigma_{f:k_1}(R) \setminus \sigma_{f:k_2}(R) = \sigma_{f:k_1 \neg f:k_2}(R) \tag{2c}$$
$$\sigma_{f:k_1}(\sigma_{f:k_2}(R)) = \sigma_{f:k_1 \wedge f:k_2}(R) \tag{2d}$$

Dataset and Entity Projection. The projection operator π allows to extract specific columns, such as *dataset* or *entity*, from a relation. For example, the expression:

$$\pi_{d,e}(R)$$

returns a relation with only two columns, dataset and entity, as shown in Figure 2b.

The projection is idempotent, a series of projections is equivalent to the outermost projection, and is distributive over set union but not over intersection and set-difference [54]:

$$\pi_e(\sigma_{f:k_1}(R_1) \cap \sigma_{f:k_2}(R_2)) \neq \pi_e(\sigma_{f:k_1}(R_1)) \cap \pi_e(\sigma_{f:k_2}(R_2))$$
$$\pi_e(\sigma_{f:k_1}(R_1) \cup \sigma_{f:k_2}(R_2)) = \pi_e(\sigma_{f:k_1}(R_1)) \cup \pi_e(\sigma_{f:k_2}(R_2))$$
$$\pi_e(\sigma_{f:k_1}(R_1) \setminus \sigma_{f:k_2}(R_2)) \neq \pi_e(\sigma_{f:k_1}(R_1)) \setminus \pi_e(\sigma_{f:k_2}(R_2))$$

4. Node-Based Indexing for Web Data*

In this section, we present the Semantic Information Retrieval Engine, SIREn, a system based on Information Retrieval (IR) techniques and conceived to index Web Data and search entities in large scale scenarios. The requirements have therefore been:

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

With respect to point 1, 2 and 3, we have developed SIREn to support the Entity Attribute-Value model from Section 3.2.4 and the boolean search model from Section 3.3, since these models cover RDF, Microformats and likely other forms of semi-structured data that can be found of the Web. Finally, we will see in Section 4.3 that the entity centric indexing enables SIREn to leverage well known Information Retrieval techniques to address the point 4.

The section is organized as follows. We present the node-labelled data model in Section 4.1 and the associated query model in Section 4.2. We describe in Section 4.3 how to extend inverted lists as well as update and query processing algorithms to support the node labelled data model. An analysis of the differences and theoretical performances between SIREn and other entity retrieval systems is given in Section 4.4.

*This section is partially based on [55]

4.1. Node-Labelled Tree Model

SIREn adopts a node-labelled tree model to capture the relation between datasets, entities, attributes and values. The tree model is pictured in Figure 6a. The tree has four different kind of nodes: dataset, entity, attribute and value. The organisation of the nodes in the tree is based on the containment order of the concepts (i.e., dataset, entity, attribute, value). Other tree layout would create unnecessary node repetitions. In fact, the tree encodes a Entity Attribute Value table, where each branch of the tree represents one row of the Entity Attribute Value table. Each node can refer to one or more terms. In the case of RDF, a term is not necessarily a word from a literal, but can be an URI or a local blank node identifier. The incoming relations of an entity, i.e., all edges where the entity node is the target, are symbolised by a attribute node with a $^{-1}$ tag in Figure 6b.

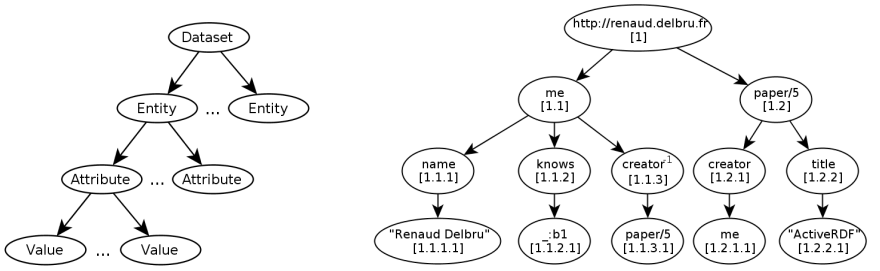
A node-labelled tree model enables one to encode and efficiently establish relationships between the nodes of a tree. The two main types of relations are *Parent-Child* and *Ancestor-Descendant* which are also core operations in XML query languages such as XPath. To support these relations, the requirement is to assign unique identifiers, called *node labels*, that encode the relationships between the nodes. Several node labelling schemes have been developed [23] but in the rest of the paper we use a simple prefix scheme, the *Dewey Order* encoding [56].

In Dewey Order encoding, 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 prefix of label(v). Figure 6b depicts a data tree where nodes have been labelled using Dewey's encoding. Given the label $\langle 1.1.1.1 \rangle$ for the term `Renaud`, we can find that its parent is the attribute `name`, labelled with $\langle 1.1.1 \rangle$.

The node labelled tree is embedded into an inverted index. The inverted index stores for each term occurrence its node label, i.e., the path of elements from the root node (dataset) to the node (attribute or value) that contains the term. A detailed description of how this tree model is encoded into an inverted index is given in Section 4.3.

4.2. Query Model

In this section, we present a set of query operators over the content and the structure of the node-labelled tree which covers the boolean search model presented



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

(b) Node-labelled tree model using Dewey's encoding of the example dataset from Section 3.1.1

Figure 6: The node-labelled tree model

in Section 3.3. We will present the query operators of SIREn and whenever possible compare them with the search query algebra from Section 3.3.1.

4.2.1. Content operators

The content operators are the only ones that access the content of a node and are orthogonal to the structure operators. The atomic search element is a keyword. Multiple keywords can be combined with traditional keyword search operations. Such operations include boolean operators (intersection, union, difference), proximity operators (phrase, near, before, after, etc.), fuzzy or wildcard operators, etc.

These operators give the ability to express complex keyword queries. A keyword query is used to retrieve a particular set of nodes. Interestingly, it is possible to apply these operators not only on literals, but also on URIs, if URIs are tokenized and normalized. For example one could just use an RDF local name, e.g., `name`, to match `foaf:name` ignoring the namespace.

With respect to the search query algebra of Section 3.3.1, the content operators are mapped to the *keyword selection* operators. The query algebra only defines the boolean operators, but it is easy to see how to extend the algebra for including proximity or other operators. Also, the content operators allow to restrict keyword search to a particular type of nodes, being either dataset, entity, attribute or value. However, in the following we assume that this restriction is implicit and thus is not shown in the following examples.

4.2.2. Structure operators

The structure operators are accessing the structure of the data tree. The atomic search element is a node. Multiple nodes can be combined using tree and set operators. The tree operators, i.e. the Ancestor-Descendant and Parent-Child operators, allow to query node relationships and to retrieve the paths matching a given pattern. The combination of paths are possible using set operators, enabling the computation of star-shaped queries such as the one pictured in Figure 5.

Ancestor-Descendant - A//D. A node A is the ancestor of a node D if there exists a path between A and D. The operator checks the node labels and retains only relations where the label of A is a prefix of the label of D. For example, in Figure 6b, the operation `renaud.delbru.fr // paper/5` will retain the relations `[1] // [1.1.3.1]` and `[1] // [1.2]`. With respect to the query algebra from Section 3.3.1, we interpret an Ancestor-Descendant operator as a keyword selection applied on a second operation. For example, Query Q4 in Appendix A can be interpreted as an Ancestor-Descendant operator where $\sigma_{d:biblio}$ is the ancestor and $R_1 \cap R_2$ is the descendant.

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. The operator checks the node labels and retains only relations where the label of P is the longest prefix matching the label of C. For example, in Figure 6b, the operation `creator`$^{-1}$ `/ paper/5` will retain the relation `[1.1.3] / [1.1.3.1]`. With respect to the query algebra from Section 3.3.1, we also interpret a Parent-Child operator as a keyword selection applied on a second operation. Query Q2 in Appendix A can be interpreted as an Parent-Child operator where $\sigma_{at:author}$ is the parent and $\sigma_{v:john \wedge v:smith}$ is the child.

Set manipulation operators. These operators allow the manipulation of nodes (dataset, entity, attribute and value) as sets, implementing union ($\cup$), difference ($\setminus$) and intersection ($\cap$). These operators are mapped one to one to the set operators found in the query algebra from Section 3.3.1. For example, Query Q3 in Appendix A can be interpreted as an intersection $R_1 \cap R_2$ between

two Parent-Child operators, $\sigma_{at:author}(\sigma_{v:john \wedge v:smith})$ and $\sigma_{at:title}(\sigma_{v:search \wedge v:engine})$.

Projection. The *dataset* and *entity* projection defined in Section 3.3.1 are simply performed by applying a filter over the node labels in order to keep the dataset and entity identifiers and filter out unnecessary identifiers such as the attribute or value identifier.

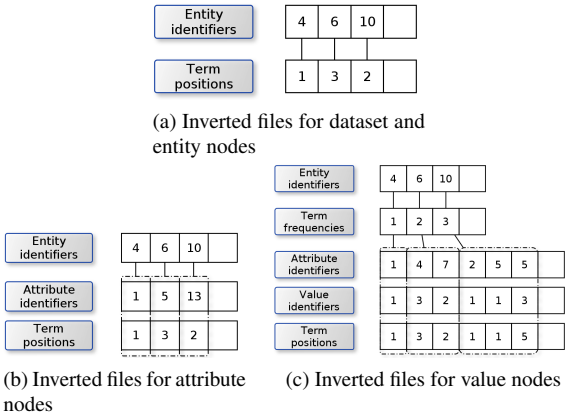
4.3. Implementing the Model

We describe in this section how the tree model is implemented on top of an inverted index. We first describe how we extend the inverted index data structure before explaining the incremental index updating and query processing algorithms. We refer the reader to [55] which discusses how query results are ranked during query processing.

4.3.1. Node-Based Inverted Index

An inverted index is composed of (1) a lexicon, i.e., a dictionary of terms that allows fast term lookup; and (2) of a set of inverted lists, one inverted list per term. In a node-based inverted index, the node labels, or Dewey's vectors, that are associated with each term are stored within the inverted lists. Compared to the traditional document-based inverted index, the difference is situated in the structure of the inverted lists. Originally, an inverted list is composed of a list of document identifiers, a list of term frequencies and a list of term positions. In our implementation, an inverted list is composed of five different streams of integers: a list of entity identifiers, of term frequencies, of attribute identifiers, of value identifiers and of term positions. The term frequency corresponds to the number of times the term has been mentioned in the entity description. The term position corresponds to the relative position of the term within the node.

However, it is unnecessary to associate each term to the five streams of integers. For example, a term appearing in an attribute node does not have a value identifier. Also, the probability to have more than one occurrence of the same term in the entity, attribute and dataset nodes is very low. Therefore, we assume that terms from entity, attribute and dataset nodes always appear a single time in the node. In that case, their term frequency is always equal to one, and it becomes unnecessary to store it. By carefully selecting what information is stored for each term, the index size is reduced and the overall performance improves since less data has to be written during indexing and read during query processing. The



(a) Inverted files for dataset and entity nodes

(b) Inverted files for attribute nodes

(c) Inverted files for value nodes

Figure 7: Diagram showing the set of inverted lists and their inter-connection for each type of terms.

strategy is to associate each term to a set of different inverted files depending on which node the term appears as described next. This depicted in Figure 7.

dataset and entity: Terms from a dataset or entity node are associated to a list of entity identifiers and to their relative position within the node as shown in Figure 7a. The position of a term within a node is necessary to support phrase and proximity queries.

attribute: Terms from an attribute node are associated to a list of entity identifiers, a list of attribute identifiers and to their relative position within the node as shown in Figure 7b.

value: Terms from a value node are associated to a list of entity identifiers, a list of term frequencies, a list of attribute identifiers, a list of value identifiers and to their relative positions within the node as shown in Figure 7c. We consider that a term can appear more than once in one or more value nodes. Therefore, each entity identifier is associated to a variable number (specified by the term frequency) of attribute identifiers, value identifiers and positions.

Instead of storing the dataset identifier of the Dewey's vector, we are encoding the relation between dataset terms and entity identifiers. This can be considered as a simplification of the data model from Section 4.1. This approach only enables a partial support of dataset queries since queries such as Query Q5 in Appendix A can not be answered efficiently. However, this reduces the update complexity and enables more efficient incremental updates. Such a choice is discussed more in details in Section 4.3.4.

4.3.2. Incremental Update of the Inverted Lists

The proposed model supports incremental updates of entities as it is performed for documents in traditional inverted indexes [57]. Adding an entity corresponds to adding a set of statements to the inverted index. The statements are first transformed into a node-labelled tree data model as in Figure 6b. Then, for each term of the tree, the associated inverted lists are accessed and updated with respect to the Dewey's vectors and positions of the term.

For example, to add a new occurrence of a term t from a value node, the following operations are performed:

1. the inverted files of t is accessed by performing a lookup in the lexicon;
2. a new entry is appended to the list of entity identifiers, term frequencies, attribute identifiers, value identifiers and positions.

If the same term t appears in a entity or dataset node, then the operations are similar than the previous ones with the difference that only the list of entity identifiers and term positions is accessed and updated.

The complexity of insertion of one term occurrence is $O(log(n) + k)$, where the term $log(n)$ denotes the cost of looking up a term in a lexicon of n terms and the term k denotes the cost of appending an integer to k inverted lists. The lexicon lookup is the predominant cost during the insertion of one term occurrence. However, updates are usually performed by batches of multiple entities. In this case, the update complexity becomes linear with the number of term occurrences since a lexicon lookup is only performed once per term.

Compared to common RDF databases, we do not support deletion on a statement granularity, but we support the deletion of a dataset or entity. When an entity is removed, its identifier is inserted into a *deletion table*. When a dataset is deleted, the associated entity identifiers are inserted into the deletion table. During query processing, each entity identifiers 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 a such maintenance operation.

4.3.3. Query Processing

The evaluation of a query works in a bottom-up fashion. First, matching on the content (terms) of a node is performed, then node information is used during list intersection for filtering the result candidates that do not belong to a same node or branch of the tree.

The intersection of two inverted lists is the most common operation during query processing. For example, a content operator such as the boolean intersection and phrase proximity operators relies on a list intersection. The structure operators such as the Ancestor-Descendant, Parent-Child and boolean intersection operators also rely on list intersection. The methodology is identical for all of them and is described by the following merge algorithm:

1. The inverted list of each term is retrieved.
2. We position the pointers to the first element of each inverted list.
3. We then walk through the inverted lists simultaneously. At each step, we perform the following operations:
   (a) We first compare their node information.
   (b) If the comparison is positive,
      i. we add the entity identifier to the result list;
      ii. we advance the list pointers to their next position.
   (c) If the comparison is negative, we move forward the list pointer with the smallest identifier to its next position.

However, the comparison between node information that is performed at step 3.a is slightly different depending on the query operator employed. In the case of a boolean intersection between two words, the algorithm compares first their entity identifiers, then their attribute identifiers and finally compare their value identifiers. In the case of a proximity operator, the position information is additionally compared. Concerning Ancestor-Descendant and Parent-Child operators, the comparison is restricted to the elements of the ancestor node to mimic the node label prefix matching as explained in Section 4.1. For example, if a word from a dataset node and a word from a value node are intersected to check a Ancestor-Descendant relation, then only the entity identifier is compared.

Given the query `creator`$^{-1}$ `/ paper/5`, the query evaluation is performed as follows. In the following examples, we display the node information as a Dewey's vector and omit the position information for simplicity.

1. Postings List Fetching
   (a) Retrieve the inverted list for the term "creator^{-1}": `[1.0]`, `[1.3]`
   (b) Retrieve the inverted list for the term "paper/5": `[1.3.1]`
2. Inverted List Merging

(a) position the pointers to the first element of the two lists.
(b) compare the entity identifiers. The two entity identifiers are equal to 1.
(c) compare the attribute identifiers. The first pointer has an attribute identifier <0> inferior to the second pointer <3>. We move the first pointer to the next occurrence.
(d) compare the attribute identifiers. This time, the two attribute identifiers are equal to 3. We have a match and add the entity identifier in the result list.

The worst-case complexity of a query evaluation is in time linear to the total number of term occurrences in the inverted list [58]. In the average case, the complexity of an intersection is reduced to sub-linear time with the use of self-indexing [59] over the entity identifiers in order to skip and avoid unnecessary record comparisons.

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 [60] enabling the use of effective merge-joins without intermediate result sorting.

4.3.4. Handling Dataset Query

As we explained previously, the current implementation partially supports dataset queries. Dataset queries such as Query Q4 from Appendix A that restrict entity matching to a certain dataset are possible. However, it is not possible to support efficiently dataset queries such as Query Q5 that retrieve all datasets involving two or more entity queries. The reason is that it is not possible at query time to know the dataset identifier associated to each entity, thus making impossible the intersection of the two entity relations based on an equality condition over their dataset identifier.

In order to support such an intersection, we would have to encode the relation between the dataset node with all the other nodes by storing a list of dataset identifiers in every inverted lists. However, this will considerably increase the update complexity since it will be necessary to keep a global order in the dataset identifier lists and a local order in the entity identifier lists within each dataset. With such requirements, it is difficult to implement an efficient incremental update procedure.

4.4. Comparison among Entity Retrieval Systems

In this section, we evaluate four entity retrieval systems: SIREn based on a node-labelled index, field-based indexes [25], RDF databases [27, 28, 29, 30] based on quad tables and Semplore [37]. These techniques are representative of the current approaches for entity retrieval. Comparing under fair conditions these techniques with an experimental benchmark is extremely difficult due to the differences in the implementations, programming languages, code optimisations, and features for which these systems are designed and built. In this work we therefore concentrate on providing a theoretical comparison. However, we refer the interested reader to [55] where we perform an experimental benchmark to compare the performance of SIREn against RDF databases with respect to incremental indexing and structured query processing. As we can see in this paper, perhaps one of the most striking difference is in the index maintenance where inverted index shows a nearly-constant update time.

A quad table is a fairly conventional data structure in RDF database mangement systems that is composed of four columns s, p, o, c called respectively "subject", "predicate", "object" and "context". In each table, quads are sorted and indexed by a particular set of columns. This allows for the quick retrieval of quads that conform to access patterns where any of these columns are specified. These access patterns are the basic building blocks from which complex SPARQL queries are constructed. In order to cover all access patterns, multiple quad tables are necessary [27].

Field-based indexing schemes are generally used in standard document retrieval systems such as Apache Lucene[3] to support basic semi-structured information like document's fields, e.g., the title. A field index is a type of sequence index scheme (see Section 2.1) that constructs lexicon terms by concatenating the field or attribute name, e.g., the predicate URI, with the terms from the content of this field. For example, in the graph depicted in Figure 3, the index terms for the entity "giovanni" and its predicate *name* will be represented as *name:giovanni* and *name:tummarello*. In fact, the field index encodes the relation between an attribute and a value directly within the lexicon.

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.

[3] Apache Lucene: `http://lucene.apache.org/`

Criteria	Node Index	Field Index	Quad Table	Semplore
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
Context	Partial	Partial	Yes	Partial
Precision (false positive)	No	Yes	No	Yes

Table 3: Summary of comparison among the four entity retrieval systems

In the following, we assume that term dictionaries as well as quad tables are implemented on top of a b+-tree data structure for fast record lookups. The comparison is performed according to the following criteria: *Processing Complexity*, *Update Complexity*, *Query Expressiveness* and *Precision*. *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. *Precision* evaluates if the system returns any false answers in the query result set.

4.4.1. Processing Complexity

Since the field-based index encodes the relation between an attribute and a value term 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 attributes. This overhead can have a significant impact on the query processing time. In contrast, the other systems has a term dictionary of size n and thus a dictionary lookup complexity of $O(log(n))$.

To lookup a quad or triple pattern, the complexity of the node and field index is equal to the complexity of looking up a few terms in the dictionary. In contrast, RDF databases have to perform an additional 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 $log(k)$ the complexity to lookup a quad in a quad table, with k being the number of quads in the database. In general, it is expected to have considerably more quads than terms, with k generally much larger than n. Therefore, the quad table lookup has a substantial impact on the query processing time for very large data collection.

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. However, this kind of join can be performed on average in sub-linear time. On the contrary, the other indexes do not have to perform such a join, since the field index encodes the relationship between predicate and object in the dictionary, the quad table in the b+-tree and Semplore in the inverted list for each term occurrence (but only for URI terms and not literal terms). Furthermore, in Semplore, access patterns where the predicate is not specified trigger a full index scan which is highly inefficient.

For evaluating a star-shaped query (joining multiples quad patterns), each index has to perform a join between the results of all the patterns. Such a join is linear with the number of results in the case of the quad table, and sub-linear in average for the node and field index with the use of the self-indexing method [59]. In contrast, Semplore has often to resort to possibly expensive external sort before merge-join operations.

4.4.2. Update Complexity

In a b+-tree system the cost of insertion of one quad represents the cost of searching the related 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 the tree. These costs become problematic with large indices and requires advanced optimizations [61] that in return cause a degradation 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 Section 4.3.2, 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, one for each access pattern, have to be maintained which limits effective caching. 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 full reindexing. The authors have recently [37] proposed an extension for incremental maintenance operations based on the *landmark* [62] technique but the update complexity remains sustained. The update cost is $O(log(n) + log(l))$ with l the number of landmarks in the inverted list. The fact that Semplore uses multiple indexes and landmarks considerably increase the update complexity. For example, index size and creation time reported in [37] are higher than for the state-of-the-art RDF database RDF-3X [30].

4.4.3. Query Expressiveness

In terms 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. Compared to field index and Semplore, the node index provides more flexibility since it enables to keyword search on every parts of a quad. In addition, node indexes support set operations on every nodes, giving the ability to express queries over multi-valued attributes. For example, a field-based index and Semplore cannot process Query Q1 from Appendix A without potentially returning false-positive results.

While Semplore supports relational, tree shaped, queries, it does not index relations between a resource and a literal. Hence, it is not possible to restrict full-text search of a literal using a predicate, e.g., queries such as `(?s, <foaf:name>, "renaud")`.

The node indexing scheme, field-based indexing scheme and Semplore only support dataset queries partially. The three systems are using an identical technique that consists of encoding the relation between dataset terms with the entity identifiers as explained in Section 4.3. This approach makes difficult the processing of the dataset query Q5.

4.4.4. Precision

The field indexing scheme encodes the relation between an attribute and and a value term in the index dictionary, but loses an important structural information: the distinction between multiple values. As a consequence, the field index may return false-positive results for Query Q1. Semplore suffers from a similar problem: it aggregates all the values of an entity, disregarding the attribute, into a single bag of words. On the contrary, the node index and the quad table are able to distinguish distinct values and do not produce wrong answers.

4.5. Conclusion on Node-Based Indexing

We presented SIREn, a node-based indexing scheme for semi-structured data. SIREn is designed for indexing very large datasets and handling the requirements of indexing and querying Web Data: constant time incremental updates and very efficient entity lookup using semi-structural queries with full text search capabilities. With respect to DBMS and IR systems, SIREn positions itself somewhere in the middle as it allows semi-structural queries while retaining many desirable IR features: single inverted index, effective caching, top-k queries and efficient distribution of processing across index shards[4].

[4] An index shard is a particular subset of the entire index.

We have described its model and implementation and have shown how it supports the entity retrieval model from Section 3. We have also compared its theoretical performance with other entity retrieval systems, and shown that the node-based indexing scheme offers a good compromise between query expressiveness, query processing and index maintenance compared to other approaches. In the next section, we show how to considerably improve the overall performance of the system by developing a high-performance compression technique which is particularly effective with respect to the node-based inverted index.

5. High-Performance Compression for Node Indexing Scheme

Inverted lists represent an important proportion of the total index size and are generally kept on disk. It is desirable to use compression techniques in order to reduce disk space usage and transfer overhead. On the one hand, efficient decompression can provide faster transfer of data from disk to memory, and therefore faster processing, since the time of fetching and decompressing a data segment is less than fetching an uncompressed form [63]. On the other hand, efficient compression can provide faster indexing since the time of compressing and writing a data segment is less than the time to write an uncompressed form. To summarise, compression is useful for saving disk space, but also to maximise IO throughput and therefore to increase the update and query throughput.

In the past years, compression techniques have focussed on CPU optimised compression algorithms [64, 65, 66, 67]. It has been shown in [65, 66, 67] that the decompression performance depends on the complexity of the execution flow of the algorithm. Algorithms that require branching conditions tend to be slower than algorithms optimised to avoid branching conditions. In fact, simplicity over complexity in compression algorithms is a key for achieving high performance. The challenge is however to obtain a high compression rate while keeping the execution flow simple.

Previous works [65, 68, 66, 69, 70, 67] have focussed solely on two factors, the compression ratio and the decompression performance, disregarding the compression performance. While decompression performance is essential for query throughput, compression performance is crucial for update throughput. We therefore propose to study compression techniques with an additional third factor, the compression performance. We show that compression performance is also dependent on an optimised execution flow.

In this section, we introduce a high-performance compression technique, the Adaptive Frame of Reference (AFOR), which provides considerable benefits with respect to the node-indexing scheme presented in the previous section. We compare our approach against a number of state-of-the-art compression techniques for inverted indexes. We perform an experimental evaluation based on three factors: indexing time, compression ratio and query processing time. We show that AFOR can achieves significant performance improvements on the indexing time and compression ratio while maintaining one of the fastest query processing time.

5.1. Background

In this section, we first introduce the block-based data structure of an inverted list which is the backbone of the compression mechanism. We then recall the delta encoding technique for inverted lists which is commonly employed before compression. Finally, we describe five compression algorithms selected for the experiments and discuss their implementation.

5.1.1. Block-Based Inverted List

We now describe the implementation of an inverted file on disk. For performance and compression efficiency, it is best to store separately each data stream of an inverted list [71]. In a non-interleaved index organisation, the inverted index is composed of five inverted files, one for each inverted lists. Each inverted file stores contiguously one type of list, and five pointers are associated to each term in the lexicon, one pointer to the beginning of the inverted list in each inverted file.

An inverted file is partitioned into blocks, each block containing a fixed number of integers as shown in Figure 8. Blocks are the basic units for writing data to and fetching data from disk, but also the basic data unit that will be compressed and decompressed. A block starts with a block header. The block header is composed of the length of the block in bytes and additional metadata information that is specific to the compression technique used. Long inverted lists are often stored across multiple blocks, starting somewhere in one block and ending somewhere in another block, while multiple small lists are often stored into a single block. For example, 16 inverted lists of 64 integers can be stored in a block of 1024 integers. We use blocks of 1024 integers in our experiments, since this was providing the best performance with respect to the CPU cache. The performance of all the compression techniques decreased with smaller block sizes.

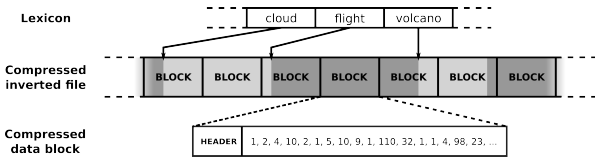


Figure 8: Inverted index structure. Each lexicon entry (term) contains a pointer to the beginning of its inverted list in the compressed inverted file. An inverted file is divided into blocks of equal size, each block containing the same number of values.

5.1.2. Delta Encoding of Inverted Lists

A more compact representation for node-based index structure is to represent node labels as delta values, a technique first introduced in [72]. The key idea of the delta compression is to store the difference between consecutive values instead of the values themselves. This allows to encode an ordered list of integers using much smaller integers, which theoretically can be encoded in less bits. In a node-based indexing scheme, the delta values are much smaller than those obtained in a document-based indexing. This is due (1) to the usually more verbose and repetitive nature of structured data, e.g., the same URI used multiple times, and (2) to the locality of the attribute identifiers, the value identifiers and the term positions. Compression techniques are then used to encode the delta values with the smallest number of bits possible.

5.1.3. Algorithms for Compressing Inverted Lists

Binary Interpolative Coding [73] has been shown to provide a very good compression rate and it could have been a reference for comparing compression rates. However, it is very inefficient in decoding, and we found that Rice is competitive enough in terms of compression rate to use it as a reference.

Rice Coding. In Rice [74], an integer n is encoded in two parts: a quotient $q = \lfloor \frac{n}{2^b} \rfloor$ and a remainder $r = n \bmod 2^b$. The quotient is stored in unary format using $q + 1$ bits while the remainder is stored in binary format using b bits. In our implementation, the parameter b is chosen per block such that 2^b is close to the average value of the block.

The main advantage of Rice is its very good compression ratio. However, it is in general the slowest method in terms of compression and decompression. The main reason is that Rice needs to manipulate the unary word one bit at a time during both compression and decompression, which is costly in CPU cycles.

Variable Byte Coding (VByte). Variable Byte compression encodes an integer with a variable number of bytes. VByte is byte-aligned, i.e., coding and decoding is done one byte at a time. Each byte consists of 7 bits to encode the partial binary representation of the integer, and one bit used as status flag to indicate if the following byte is part of the current number.

The advantages of VByte are: (1) it is simple to implement; and (2) its overall compression and decompression performance are good. Compared to bitwise techniques like Rice, VByte requires a single branching condition for each byte which is more CPU cost-effective. However, the branching condition leads to branch mispredictions which makes it slower than CPU optimised techniques such as the one presented next. Moreover, VByte has a poor compression ratio since it requires one full byte to encode one small integer (i.e., $\forall n | n < 2^7$).

Simple Coding Family. The idea behind the Simple coding is to pack as many integers as possible into one machine word (being 32 or 64 bits). We describe one Simple coding method (referred to as S-64 in our experiments) based on 64-bit machine words, recently introduced in [67]. In our experiments, we report only S-64 results since its performance was always superior to Simple9 [65]. In S-64, each word consists of 4 status bits and 60 data bits. The 4 status bits are used to encode one of the 16 possible configurations for the data bits. A description of the 16 configurations can be found in [67]. S-64 wastes generally less bits than Simple9 and therefore provides a better compression ratio.

In addition to providing good compression ratio, decompression is done efficiently by reading one machine word at a time and by using a precomputed lookup table over the status bits in order to select an optimised routine (one routine per configuration) to decode the data bits using shift and mask operations only. However, one disadvantage is that compression cannot be done efficiently. The typical implementation is to use a sliding window over the stream of integers and to find the best configuration, i.e., the one providing the best compression ratio, for the current window. This generally requires repetitive try and error iterations over the possible configurations for each new window. In addition, Simple coding has to perform one table lookup per machine word and consumes more CPU cycles than the techniques presented next.

Frame of Reference (FOR). FOR determines the range of possible values in a block, called a *frame*, and maps each value into this range by storing just enough bits to distinguish between the values [64]. Given a frame $[min, max]$, FOR needs $\lceil \log_2(max - min + 1) \rceil$ bits, that we call *bit frame* in the rest of the paper, to encode each integer in a block. In the case of the delta-encoded list of values, since the probability distribution generated by taking the delta tends to be naturally monotonically decreasing, one common practice [66, 67] is to choose as frame the range $[0, max]$ where max is the largest number in the group of delta values.[5]

The main disadvantage of FOR is that it is sensitive to outliers in the group of values. For example, if a block of 1024 integers contains 1023 integers inferior to 16, and one value superior to 128, then the bit frame will be $\lceil \log_2(128 + 1) \rceil = 8$, wasting 4 bits for each other values. However, compression and decompression is done very efficiently using highly-optimised routines [66] which avoid branching conditions. Each routine is loop-unrolled to encode or decode m values using shift and mask operations only. Listing 1 and 2 show the routines to encode or decode 8 integers with a bit frame of 3. There is a compression and decompression routine for each bit frame.

Given a block of n integers, FOR determines a frame of reference for the block and encodes the block by small iterations of m integers using the same compression routine at each iteration. Usually, and for questions of performance, m is chosen to be a multiple of 8 so that the routines match byte boundaries. In our implementation, FOR relies on routines to encode and decode 32 values at a time.

The selection of the appropriate routine for a given bit frame is done using a precomputed lookup table. The compression step performs one pass only over the block to determine the bit frame. Then, it selects the routine associated to the bit frame using the lookup table. Finally, the bit frame is stored using one byte in the block header and the compression routine is executed to encode the block. During decompression, FOR reads the bit frame, performs one table lookup to select the decompression routine and executes iteratively the routine over the compressed block.

Patched Frame Of Reference (PFOR). PFOR [66] is an extension of FOR that is less vulnerable to outliers in the value distribution. PFOR stores outliers as exceptions such that the frame of reference $[0, max]$ is greatly reduced. PFOR first determines the smallest max value

[5] This assumes that a group of values will always contain 0, which is not always the case. However, we found that taking the real range $[min, max]$ was only reducing the index size by 0.007% while increasing the complexity of the algorithm.

```
encode3(int[] i, byte[] b)
  b[0] = (i[0] & 7)
       | ((i[1] & 7) << 3)
       | ((i[2] & 3) << 6);
  b[1] = ((i[2] >> 2) & 1)
       | ((i[3] & 7) << 1)
       | ((i[4] & 7) << 4)
       | ((i[5] & 1) << 7);
  b[2] = ((i[5] >> 1) & 3)
       | ((i[6] & 7) << 2)
       | ((i[7] & 7) << 5);
```

Listing 1: Loop unrolled compression routine that encodes 8 integers using 3 bits each

```
decode3(byte[] b, int[] i)
  i[0] = (b[0] & 7);
  i[1] = (b[0] >> 3) & 7;
  i[2] = ((b[1] & 1) << 2)
       | (b[0] >> 6);
  i[3] = (b[1] >> 1) & 7;
  i[4] = (b[1] >> 4) & 7;
  i[5] = ((b[2] & 3) << 1)
       | (b[1] >> 7);
  i[6] = (b[2] >> 2) & 7;
  i[7] = (b[2] >> 5) & 7;
```

Listing 2: Loop unrolled decompression routine that decodes 8 integers represented by 3 bits each

such that the best compression ratio is achieved based on an estimated size of the frame and of the exceptions. Compressed blocks are divided in two: one section where the values are stored using FOR, a second section where the exceptions, i.e., all values superior to *max*, are encoded using 8, 16 or 32 bits. The unused slots of the exceptions in the first section are used to store the offset of the next exceptions in order to keep a linked list of exception offsets. In the case where the unused slot is not large enough to store the offset of the next exceptions, a *compulsive exception* [66] is created. Instead, we use the non-compulsive approach proposed in [70], where the exceptions are stored along with their offset in the second block section, since it has been shown to provide better performance. During our experimentations, we tried PFOR with frames of 1024, 128 and 32 values. With smaller frames, the compression rate was slightly better. However, the query time performance was decreasing. We therefore decided to use PFOR with frames of 1024 values as it was providing a good reference for query time.

The decompression is performed efficiently in two phases. First, the list of values are decoded using the FOR routines. Then, the list of values is *patched* by: (1) decompressing the exceptions and their offsets and (2) replacing in the list the exception values. However, the compression phase cannot be efficiently implemented. The main reason is that PFOR requires a complex heuristic that require multiple passes over the values of a block in order to find the frame and the set of exceptions providing the highest compression.

Vector of Split Encoding. At the time of the writing, we discovered the Vector of Split Encoding (VSE) [75] which is similar to AFOR. The two methods can be considered as an extension of FOR which are less sensitive to outliers by adapting their encoding to the value distribution. To achieve this, the two methods are encoding a list of values by partitioning it into frames of variable lengths and rely on algorithms to automatically find the list partitioning. AFOR relies on a local optimisation algorithm for partitioning a list, while VSE adds a Dynamic Programming method to find the optimal partitioning of a list.

For the purpose of our comparison, we use AFOR to show how adaptive techniques can be peculiarly effective with respect to node-based indexes. Given its almost identical nature, we can expect the results to be very close to those of a VSE implementation.

5.2. Adaptive Frame of Reference

Adaptive Frame Of Reference (AFOR) attempts to retain the best of FOR, i.e., a very efficient compression and decompression using highly-optimised routines, while providing a better tolerance against outliers and therefore achieving a higher compression ratio. Compared to PFOR, AFOR does not rely on the encoding of exceptions in the presence of outliers. Instead, AFOR partitions a block into multiple frames of variable length, the partition and the length of the frames being chosen appropriately in order to adapt the encoding to the value distribution.

To elaborate, AFOR works as follow. Given a block B of n integers, AFOR partitions it into m distinct frames and encodes each frame using highly-optimised routines. Each frame is independent from each other, i.e., each one has its own *bit frame*, and each one encodes a variable number of values. This is depicted in Figure 9 by *AFOR-2*. Along with each frame, AFOR encodes the associated bit frame with respect to a given encoder, e.g., a binary encoder. In fact, AFOR encodes (resp., decodes) a block of values by:

1. encoding (resp., decoding) the bit frame;
2. selecting the compression (resp., decompression) routine associated to the bit frame;
3. encoding (resp., decoding) the frame using the selected routine.

Finding the right partitioning, i.e., the optimal configuration of frames and frame lengths per block, is essential for achieving high compression ratio [75]. If a frame is too large, the encoding becomes more sensitive to outliers and wastes bits by using an inappropriate bit frame for all the other integers. On the contrary, if the frames are too small, the encoding wastes too much space due to the overhead of storing a larger number of bit frames. Also, alternating between large and small frames is not only important for achieving high compression ratio but

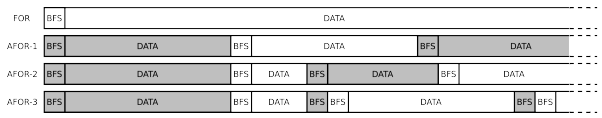


Figure 9: Comparison of block compression between FOR and AFOR. We alternate colours to differentiate frames. AFOR-1 denotes a first implementation of AFOR using a fixed frame length. AFOR-2 denotes a second implementation of AFOR using variable frame lengths. AFOR-3 denotes a third implementation using variable frame lengths and the frame stripping technique. *BFS* denotes the byte storing the bit frame selector associated to the next frame.

also for achieving high performance. If frames are too small, the system has to perform more table lookups to select the appropriate routine associated to each frame, and as a consequence the compression and decompression performance decrease. The appropriate strategy is to rely on large frames in the presence of a dense sequence of values, and on small frames in the presence of sparse sequence of values. To find a block partitioning, our solution uses a local optimisation algorithm which is explained next.

5.2.1. Partitioning Blocks into Variable Frames

Finding the optimal configuration of frames and frame lengths for a block of values is a combinatorial problem. For example, with three different frame lengths (32, 16 and 8) and a block of size 1024, there are 1.18×10^{30} possible combinations. While such a combinatorial problem can be solved via Dynamic Programming algorithms [75], the complexity of such algorithms is still $O(n \times k)$, with the term n being the number of integers and the term k the size of the larger frame, and therefore greatly impacts the compression performance. We remind the reader that we are interested not only by fast decompression speed and high compression ratio, but also by fast compression speed. Therefore, in our experiments, we do not rely on the optimal configuration. Instead, we use a local optimisation algorithm that provides a satisfactory compression rate and that is efficient to compute.

AFOR computes the block partitioning by using a sliding window over a block and determines the optimal configuration of frames and frame lengths for the current window. Given a window of size w and a list of possible frame lengths, we compute beforehand the possible configurations. For example, for a window size of 32 and three different frame lengths, 32, 16 and 8, there are six configurations: $[32]$, $[16, 16]$, $[16, 8, 8]$, $[8, 16, 8]$, $[8, 8, 16]$, $[8, 8, 8, 8]$. The size of the window as well as the number of possible frame lengths are generally chosen to be small in order to reduce the number of possible configurations. Then, we first compute the bit frames of the smallest frames by doing one pass over the values of the window as shown in the Algorithm 1 (lines 1-5). On the previous example, this means that we compute the bit frames for the configuration $[8, 8, 8, 8]$. The `bitFrames` array stores the bit frame for each of frame of this configuration. Given these bit frames, we are able to compute the bit frames of all the other frames. The second step, lines 6-12 in Algorithm 1, iterates over the possible configurations and estimates the size of each configuration in order to find the optimal one for the current window. Given the previously computed `bitFrames` array, the `EstimateSize` function computes the cost of encoding the window for a given configuration, accounting also the overhead of storing the bit frames. For example, for the configuration $[8, 8, 8, 8]$ with four frames of size 8 each, and with four associated bit frames, b_1 to b_4, the size of the encoding is computed as follow: $(4 \times 8) + 8 \times \sum_{i=1...4} b_i$, where $8 \times \sum_{i=1...4} b_i$ is the size (in bits) of the four encoded frames and 4×8 is the overhead (in bits) to store the four bit frames.

This simple algorithm is efficient to compute, in particular if the window size is small and if there is a few number of possible frame lengths. However, it is easy to see that such a method does not provide the optimal configuration for a complete block. There is a trade-off between optimal partitioning and complexity of the algorithm. One can possibly use a more complex method for achieving a higher compression if the compression speed is not critical. However, this is not the case for a web search engine where high update throughput is crucial. We decided to use this method since in our experiments we found that a small window size of 32 values and three frame lengths, 32, 16 and 8, were providing satisfactory results in terms of compression speed and compression ratio. More details about our implementations of AFOR are given next.

5.2.2. Frame Stripping

In an inverted list, it is common to encounter a long sequence of 1 to encode. For example, this occurs with terms that appear frequently in many entities. With RDF data, such a very common term might be a predicate URI or a ubiquitous class URI. As a consequence, the list of entity identifiers is composed of many consecutive identifiers, which is encoded as a list of 1 using the delta representation. Also, the schema used across the entity descriptions coming from a same dataset is

Algorithm 1: The algorithm that finds the best configuration of frames and frame lengths for a window W of size w.

```
input  : A window W of size w
input  : The smallest frame length l
output: The best configuration for the window

for i ← 0 to w/l do
    for j ← i × l to (i + 1) × l do
        bitFrames[i]
        ← max(bitFrames[i], ⌈log2(W[j] + 1)⌉);
    end
end
bestSize ← MaxSize;
foreach configuration c of the possible configurations do
    if EstimateSize(c, bitFrames) < bestSize then
        bestSize ← EstimateSize(c);
        bestConf ← c;
    end
end
```

generally similar. When indexing batch of entities coming from a same dataset, we benefit from a "term clustering" effect: all the schema terms are associated with long runs of consecutive entity identifiers in the inverted index. There is also other cases where a long run of 1 is common, for example in:

- the list of term frequencies for terms that appear frequently a single time in the entity description, e.g., class URIs;
- the list of value identifiers for terms that appear frequently in single-valued attributes;
- the list of term positions for nodes holding a single term, e.g., URIs.

In presence of such long runs of 1, AFOR still needs to encode each value using 1 bit. For example, a frame of 32 values will encode a sequence of 1 using 32 bits. The goal of the *Frame Stripping* method is to avoid the encoding of such frames. Our solution is to *strip* the content of a frame if and only if the frame is exclusively composed of 1. We encode such a case using a special bit frame.

5.2.3. Implementation

We present three different implementations of the AFOR encoder class. We can obtain many variations of AFOR by using various sets of frame lengths and different parameters for the partitioning algorithm. We tried many of them during our experimentation and report here only the ones that are promising and interesting to compare.

AFOR-1. The first implementation of AFOR, referred to as AFOR-1 and depicted in Figure 9, is using a single frame length of 32 values. To clarify, this approach is identical to FOR applied on small blocks of 32 integers. This first implementation shows the benefits of using short frames instead of long frames of 1024 values as in our original FOR implementation. In addition, AFOR-1 is used to compare and judge the benefits provided by AFOR-2, the second implementation using variable frame lengths. Considering that, with a fixed frame length, a block is always partitioned in the same manner, AFOR-1 does not rely on the partitioning algorithm presented previously.

AFOR-2. The second implementation, referred to as AFOR-2 and depicted in Figure 9, relies on three frame lengths: 32, 16 and 8. We found that these three frame lengths give the best balance between performance and compression ratio. Additional frame lengths were rarely selected and the performance decreased due to the larger number of partitioning configurations to compute. Reducing the number of possible frame lengths was providing slightly better performance but slightly worse compression ratio. There is a trade-off between performance and compression effectiveness when choosing the right set of frame lengths. Our implementation relies on the partitioning algorithm presented earlier, using a window size of 32 values and six partitioning configurations [32], [16, 16], [16, 8, 8], [8, 16, 8], [8, 8, 16], [8, 8, 8, 8].

AFOR-3. The third implementation, referred to as AFOR-3 and depicted in Figure 9, is identical to AFOR-2 but employs the frame stripping technique. Compared to AFOR-2, the compressed block can contain frames encoded by a single bit frame as depicted in Figure 9. AFOR-3 implementation relies on the same partitioning algorithm as AFOR-2 with an additional step to find and strip frames composed of a sequence of 1 in the partitions.

Compression and decompression routines. Our implementations rely on highly-optimised routines such as the ones presented in Listing 1 and 2, where each routine is loop-unrolled to encode or decode a fixed number of values using shift and mask operations only. There is one routine per bit frame and per frame length. For example, for a frame length of 8 values, the routine encodes 8 values using 3 bits each as shown in Listing 1,

while for a frame length of 32, the routine encodes 32 values using 3 bits each.

Since AFOR-1 uses a single frame length, it only needs 32 routines for compression and 32 routines for decompression, i.e., one routine per bit frame (1 to 32). With respect to AFOR-2, since it relies on three different frame lengths, it needs 96 routines for compression and 96 routines for decompression. With respect to AFOR-3, one additional routine for handling a sequence of 1 is added per frame length. The associated compression routine is empty and does nothing since the content of the frame is not encoded. Therefore the cost is reduced to a single function call. The decompression routine consists of returning an array of 1. Such routines are very fast to execute since there are no shift or mask operations.

Bit frame encoding. Recall that the bit frame is encoded along with the frame, so that, at decompression time, the decoder can read the bit frame and select the appropriate routine to decode the frame. In the case of AFOR-1, the bit frame varies between 1 to 32. For AFOR-2, there are 96 cases to be encoded, where cases 1 to 32 refer to the bit frames for a frame length of 8, cases 33 to 63 for a frame length of 16, and cases 64 to 96 for a frame length of 32. In AFOR-3, we encode one additional case per frame length with respect to the frame stripping method. Therefore, there is a total of 99 cases to be encoded. The cases 97 to 99 refer to a sequence of 1 for a frame length of 8, 16 and 32 respectively.

In our implementation, the bit frame is encoded using one byte. While this approach wastes some bits each time a bit frame is stored, more precisely 3 bits for AFOR-1 and 1 bits for AFOR-2 and AFOR-3, the choice is again for a question of efficiency. Since bit frames and frames are interleaved in the block, storing the bit frame using one full byte enables the frame to be aligned with the start and end of a byte boundary. Another implementation to avoid wasting bits is to pack all the bit frames at the end of the block. We tried this approach and report that it provides slightly better compression ratio, but slightly worse performance. Since the interleaved approach was providing better performance, we decided to use it in our experiment.

Routine selection. A precomputed lookup table is used by the encoder and decoder to quickly select the appropriate routine given a bit frame. Compared to AFOR-1, AFOR-2 and AFOR-3 have to perform more table lookups for selecting routines since they are likely to rely on small frames of 8 or 16 values when the value distribution is sparse. While these lookups cost additional CPU cycles, we will see in the experiments that the overhead is minimal.

5.3. Experiments

This section describes the benchmark experiments which aim to compare the techniques introduced by AFOR with the compression methods described in Section 5.1.3. The first experiment measures the indexing performance based on two aspects: (1) the indexing time; and (2) the index size. The second experiment compares the query execution performance.

Experimental Settings. The hardware system we use in our experiments is a 2 x Opteron 250 @ 2.4 GHz (2 cores, 1024 KB of cache size each) with 4GB memory and a local 7200 RPM SATA disk. The operating system is a 64-bit Linux 2.6.31-20-server. The version of the Java Virtual Machine (JVM) used during our benchmarks is 1.6.0_20. The compression algorithms and the benchmark platform are written in Java and based on the open-source project Apache Lucene[6].

Experimental Design. Each measurement was made by (1) flushing the OS cache; (2) initialising a new JVM and (3) warming the JVM by executing a certain number of times the benchmark. The JVM warmup is necessary in order to be sure that the OS and the JVM have reached a steady state of performance, e.g., that the critical portion of code is JIT compiled by the JVM. The implementation of our benchmark platform is based on the technical advice from [76], where more details about the technical aspects can be found.

Data Collection. We use three real web datasets for our comparison:

Geonames: a geographical database and contains 13.8 million of entities[7]. The size is 1.8GB compressed.

DBPedia: a semi-structured version of Wikipedia and contains 17.7 million of entities[8]. The size is 1.5GB compressed.

Sindice: a sample of the data collection currently indexed by Sindice. There is a total of 130,540,675 entities. The size is 6.9GB compressed.

We extracted the entity descriptions from each dataset as pictured in Figure 3.

[6] Apache Lucene: `http://lucene.apache.org/`

[7] Geonames: `http://www.geonames.org/`

[8] DBpedia: `http://dbpedia.org/`

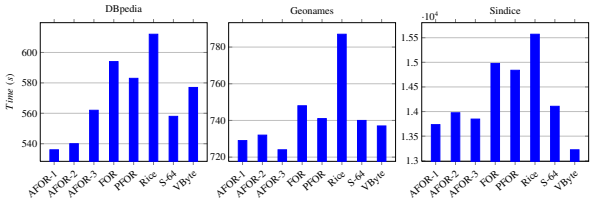


Figure 10: The total time spent to commit batches of 10000 document.

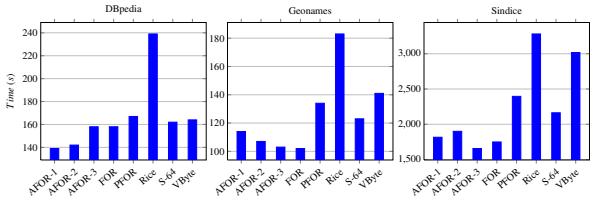


Figure 11: The total time spent to optimise the complete index.

5.3.1. Indexing Performance

The performance of indexing is compared based on the index size (compression ratio), commit time (compression speed) and optimise time (compression and decompression speed). The indexing is performed by adding incrementally 10000 documents at a time (which is similar to what others have done [77]) and finally by optimising the index. For each batch of documents, the commit operation creates a small inverted index (called an *index segment*). The optimisation merges all the index segments into a single segment. We believe this to be a common operation for incremental inverted index.

We report the results of the indexing experiments in Table C.5. The table comprises two columns with respect to the indexing time: the total commit time (*Total*) to add all the documents and the optimisation time (*Opt*). The time collected is the CPU time used by the current thread and comprises the user time and the system time. The index size in Table C.5 is studied based on the size of the individual inverted file (entity, frequency, attribute, value and position) and on the total index size (by summing the size of the five inverted files). We also provide bar plots to visualise better the differences between the techniques.

Commit Time. Figure 10 shows the total time spent by each method. As might be expected, Rice is the slowest method due to its execution flow complexity. It is followed by FOR and PFOR. We can notice the inefficiency of PFOR in terms of compression speed. On a large dataset (Sindice), VByte is the best-performing method while AFOR-1, AFOR-2, AFOR-3 and S-64 provide a similar commit time. On DBpedia, AFOR-1 and AFOR-2 are the best performing methods. On Geonames, AFOR-1, AFOR-2 and AFOR-3 are the best performing methods, while S-64 and VByte perform similarly. On smaller datasets (DBpedia and Geonames), VByte, FOR and PFOR perform similarly.

Optimisation Time. Figure 11 shows the optimise time for each methods. The time to perform the optimisation step is quite different due to the nature of the operation. The optimisation operation has to read and decompress all the index segments and compress them back into a single segment. Therefore, decompression performance is also an important factor, and algorithms having good decompression speed becomes more competitive. For example, while FOR and PFOR was performing similarly to Rice in terms of indexing time on the Sindice dataset, it is ahead of Rice in terms of optimisation time. Rice is penalised by its low decompression speed. Similarly, S-64 provides close or even better optimisation performance than VByte due to its faster decompression. On smaller datasets (DBpedia and Geonames), VByte is performing well due to its good compression and decompression speed. However, we can notice on a large dataset (Sindice) the compression ratio and the decompression speed of VByte incur a large overhead. The best-performing methods are AFOR-1, AFOR-2 and AFOR-3, with AFOR-3 performing better on large datasets. The AFOR techniques take the advantage due their optimised compression and decompression routines and their good compression rate. AFOR-3 is even twice as fast as Rice on the Sindice dataset.

Compression Ratio. Figure 12 shows the total index size achieved by each method. We can clearly see the inefficiency of the VByte approach. While VByte performs generally better than FOR on traditional document-centric inverted indexes, this is not true for inverted indexes based on a node indexing scheme. VByte is not adapted to such an index due to the properties of the delta-encoded lists of values. Apart from the entity file, the values are generally very small and the outliers are rare. In that case, VByte is penalized by its inability of encoding a small integer in less than a byte. On the contrary, FOR is able to encode many small integers in one byte. Also, while PFOR is less sensitive to outliers than FOR, the gain of compression rate provided by PFOR is minimal since outliers are more rare than in traditional inverted indexes. In contrast, AFOR and S-64 are able to better adapt the encoding to the

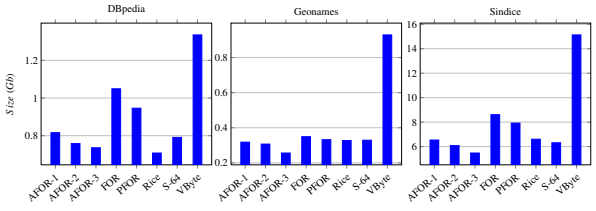


Figure 12: The index size achieved by each compression technique.

value distribution and therefore provide a better compression rate. AFOR is even able to provide better compression ratio than Rice on the Geonames and Sindice dataset. Compared to AFOR-2, we can observe in Table C.5 that AFOR-3 provides better compression rate on the frequency, value and position files, and slightly better on the entity file. This result corroborates the existence of long runs of 1 in these files, as explained in Section 5.2.2.

Conclusion on Indexing Performance. The indexing experiment shows that the compression speed is also an important factor to take into consideration when designing a compression algorithm for an inverted index. Without a good compression speed, the update throughput of the index is limited. Also the experiment shows that the optimisation operation is dependent of the decompression performance, and its execution time can double without a good compression and decompression speed. With respect to index optimisation, the compression ratio must also be taken into consideration. While VByte provides in general correct commit times, we can observe on a large dataset (Sindice) that its performance during optimisation is limited by its poor compression ratio. Overall, the method providing the best balance between indexing time, optimise time and compression ratio is AFOR. AFOR-1 provides fast compression speed and better compression ratio than FOR and PFOR. AFOR-2 provides a notable additional gain in compression ratio and optimise time but undergoes a slight increase of indexing time. AFOR-3 provides another additional gain in compression ratio while providing better compression speed than AFOR-2.

5.3.2. Query Processing Performance

We now compare the decompression performance in real settings, where inverted indexes are answering queries of various complexities. We focus on two main classes of queries, the value and attribute queries, which are the core elements of a star-shaped query. Among these two classes, we identify types of keyword queries which represent the common queries received by a web search engine: conjunction, disjunction and phrase. Under our model, these two classes of queries are the most complex ones and therefore cover the other cases (e.g., selection of a dataset or entity name and of a value).

Query Generation. The queries are generated based on the selectivity of the words composing them. The word selectivity determines how many entities match a given keyword. The words are grouped into three selectivity ranges: high, medium and low. We differentiate also two groups of words based on their position in the data graph: attribute and value. We follow the technique described in [78] to obtained the ranges of each word group. We first order the words by their descending frequency, and then take the first k words whose cumulative frequency is 90% of all word occurrences as high range. The medium range accounts for the next 10%, and the low range is composed of all the remaining words. For the phrase queries, we follow a similar technique. We first extract all the 2-gram and 3-gram[9] from the data collection. We then compute their frequency and sort them by descending frequency. We finally create the three ranges as explained above.

Value Queries Value queries are divided into three types of keyword queries: conjunction, disjunction and phrase queries. These queries are restricted to match within one single value, similar to Query Q1. Therefore, the processing of conjunction and disjunction queries relies on the entity, frequency, attribute and value inverted files. Phrase queries rely on one additional inverted file, the position inverted file.

Conjunction and disjunction queries are generated by taking random keywords from the high range group of words. 2-AND and 2-OR (resp. 4-AND and 4-OR) denotes conjunction and disjunction queries with 2 random keywords (resp. 4 random keywords). Similarly, a phrase query is generated by taking random n-grams from the high range group. 2-Phrase (resp. 3-Phrase) denotes phrase queries with 2-gram (resp. 3-gram). Benchmarks involving queries with words from low and medium ranges are not reported here for questions of space, but the performance results are comparable with the ones presented here.

Attribute Queries An attribute query is generated by associating one attribute keyword with one value query. An attribute keyword is randomly chosen from

[9] A n-gram is n words that appear contiguously

the high range groups of attribute words. The associated value query is obtained as explained previously. An attribute query intersects the result of a value query with an attribute keyword.

Query Benchmark Design. For each type of query, we (1) generate a set of 200 random queries which is reused for all the compression methods, and (2) perform 100 measurements. Each measurement is made by performing n times the query execution of the 200 random queries, with n chosen so that the runtime is long enough to minimise the time precision error of the OS and machine (which can be 1 to 10 milliseconds) to a maximum of 1%. All measurements are made using *warm cache*, i.e., the part of the index read during query processing is fully loaded in memory. The measurement time is the CPU time, i.e., user time and system time, used by the current thread to process the 200 random queries.

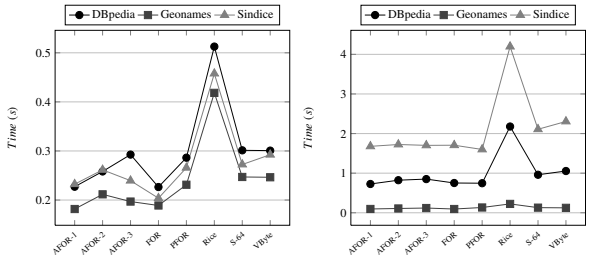
Query execution time is sensitive to external events which can affect the final execution time recorded. For instance, background system maintenance or interruptions as well as cache misses or system exceptions can occur and perturb the measurements. All these events are unpredictable and must be treated as noise. Therefore, we need to quantify the accuracy of our measurements. As recommended in [79], we report the arithmetic mean and the standard deviation of the 100 measurements. The design of the value and attribute query benchmarks includes three factors:

Algorithm having height levels: AFOR-1, AFOR-2, AFOR-3, FOR, PFOR, Rice, S-64, and VByte;
Query having six levels: 2-AND, 2-OR, 4-AND, 4-OR, 2-Phrase, and 3-Phrase; and
Dataset having three levels: DBpedia, Geonames and Sindice.

Each condition of the design, e.g., AFOR-1 / 2-AND / WIKIPEDIA, contains 100 separate measurements.

Query Benchmark Results. We report the results of the query benchmarks in Table B.4a and Table B.4b for the value and attribute queries respectively. Based on these results, we derive multiple graphical charts to better visualise the differences between each algorithm. These charts are then used to compare and discuss the performances of each algorithm.

Figure 13 and Figure 14 report the sum of the average processing time of the boolean and phrase query levels for the value and attribute queries respectively. Figure 13a and Figure 14a depict the sum of the average processing time of the boolean query levels (2-AND, 2-OR, 4-AND, 4-OR). Figure 13b and Figure 14b depict the sum of the average processing time of the phrase query level (2-Phrase, 3-Phrase). The query processing time are obtained by summing up the average time of each query from Table B.4a for the value queries and Table B.4b for the attribute queries. For example, the processing time of AFOR-1 on the DBpedia dataset in Figure 13b is obtained by summing up the processing times of the queries 2-Phrase (43.2 ms) and 3-Phrase (32.6 ms) reported in Table B.4a.



(a) Boolean Query (b) Phrase Query

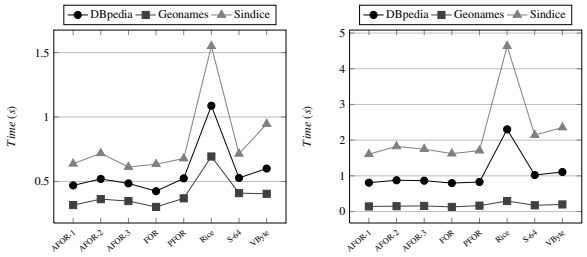
Figure 13: The sum of the average processing time of the boolean query levels (2-AND, 2-OR, 4-AND, 4-OR) for the value queries that is achieved by each compression technique.

Value Query In Figure 13, and in particular on the Sindice dataset (large dataset), we can distinguish three classes of algorithms: the techniques based on FOR, a group composed of S-64 and VByte, and finally Rice. The FOR group achieves relatively similar results, with AFOR-2 slightly behind the others.

Rice has the worst performance for every query and dataset, followed by VByte. However, Rice performs in many cases twice as slow as VByte. In Figure 13a, S-64 provides similar performance to VByte on boolean queries but we can see in Figure 13b that it is faster than VByte on phrase queries. However, S-64 stays behind FOR, PFOR and AFOR in all the cases.

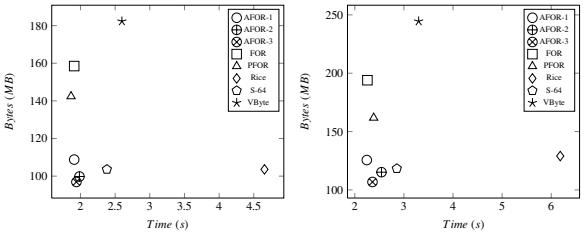
FOR, PFOR and AFOR have relatively similar performances on all the boolean queries and all the datasets. PFOR seems to provide generally slightly better performance on the phrase queries but seems to be slower on boolean queries.

Attribute Query In Figure 14, and in particular on the Sindice dataset (large dataset), we can again distinguish the same three classes of algorithms. However,



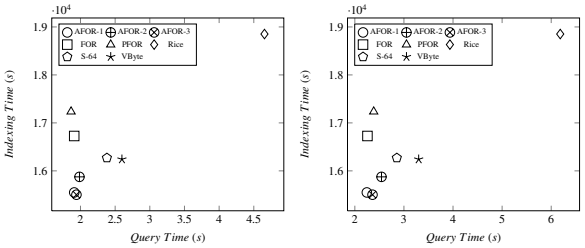
(a) Boolean Query (b) Phrase Query

Figure 14: The sum of the average processing time of the phrase query levels (2-Phrase, 4-Phrase) for the attribute queries that is achieved by each compression technique.



(a) Value Query (b) Attribute Query

Figure 15: A graphical comparison showing the trade-off between querying time and compression ratio on the Sindice dataset. The compression ratio is represented by the number of bytes read during the query processing.



(a) Value Query (b) Attribute Query

Figure 16: A graphical comparison of the compression techniques showing the trade-off between querying time and indexing time on the Sindice dataset.

the performance gap between S-64 and VByte becomes larger.

Rice has again the worst performance for every query and dataset. Compared to the performance on value queries, we can see in Figure 14a that S-64 provides similar performance to PFOR and AFOR-2 on boolean queries. FOR and AFOR-3 seem to be the best performing methods on boolean queries. With respect to the phrase queries in Figure 14b, S-64 has better performance than VByte. However, PFOR does not achieve any more the best performance on phrase queries. Instead, it seems that AFOR-2 and FOR achieve a slightly better processing time.

FOR, PFOR and AFOR have again relatively similar performances on all the queries and all the datasets. AFOR-2 appears to be slower to some degree, while the gap between AFOR-3 and PFOR becomes less perceptible.

5.3.3. Performance Trade-Off

We report in Figure 15 the trade-off between the total query processing time and the compression ratio among all the techniques on the Sindice dataset. The total query time has been obtained by summing up the average time of all the queries. The compression ratio is based on the number of bytes read during query processing which are reported in Table B.4a and Table B.4b.

We can distinctively see that the AFOR techniques are close to Rice in terms of compression ratio, while being relatively close to FOR and PFOR in terms of query processing time. Compared to AFOR-1, AFOR-2 achieves a better compression rate in exchange of a slightly slower processing time. However, AFOR-3 accomplishes a better compression rate with a very close processing time to AFOR-1.

We report in Figure 16 the trade-off between the total query processing time and the indexing time among all the techniques on the Sindice dataset. The indexing time has been obtained by summing up the commit and optimise time from Table C.5. We can distinctively see that the AFOR techniques achieve the best trade-off between indexing and querying time. AFOR-3 produce very similar indexing and querying times to AFOR-1, while providing a much better compression rate. It is interesting to notice that PFOR provides a slightly better querying time than FOR but at the price of a much slower compression. Also, S-64 and VByte provide a relatively close performance trade-off. To conclude, AFOR-3 seems to offer the best compromise between querying time, indexing time, and compression rate.

5.4. Discussion

In general, even if FOR has more data to read and decompress, it still provides one of the best query execution time. The reason is that our experiments are performed using warm cache. We therefore ignore the cost

of disk IO accesses and measure exclusively the decompression efficiency of the methods. With a cold cache, i.e., when IO disk accesses have to be performed, we expect a drop of performance for algorithms with a low compression ratio such as FOR and PFOR compared to AFOR-2 and AFOR-3. Future work will investigate this aspect.

Compression and decompression performance do not only depend on the compression ratio, but also on the execution flow of the algorithm and on the number of cycles needed to compress or decompress an integer. Therefore, CPU-optimised algorithms which provides at the same time a good compression ratio are more likely to increase the update and query throughputs of web search engines. In that context, AFOR seems to be a good candidate since it is well balanced in all aspects: it provides very good indexing and querying performance and one of the best compression ratio.

The Simple encoding family is somehow similar to AFOR. At each iteration, S-64 encodes or decodes a variable number of integers using CPU optimised routines. AFOR is however not tied to the size of a machine word, is simpler to implement and provides better compression ratio, compression speed and decompression speed.

Another interesting property of AFOR which is not discussed in this paper is its ability to skip quickly over chunks of data without having to decode them. This is not possible with techniques such as Rice, VByte or PFOR. AFOR has to decode the bit frame to know the length of the following frame, and is therefore able to deduce the position of the next bit frame. Such a characteristic could be leveraged to simplify the self-indexing of inverted files [59].

5.5. Conclusion on High-Performance Compression

We presented AFOR, a novel class of compression techniques for inverted lists which provide tremendous benefit in the case of a node-based inverted index. AFOR is specifically designed to increase update and query throughput of web search engines. We have described three different implementations of the AFOR encoder class. We have compared AFOR to alternative approaches, and have shown experimental evidences that AFOR provides a well balanced trade-off between three factors: indexing time, querying time and compression ratio. In particular, AFOR-3 achieves similar query processing times than FOR and PFOR, a better compression rate than Rice and the best indexing times.

The results of the experiment lead to interesting conclusions. In particular and with respect to the node-based indexing scheme, we have shown (1) that VByte is inadequate due to its incapacity of efficiently encoding small values; and (2) that PFOR provides only limited benefits compared to FOR. On the contrary, techniques such as S-64 and AFOR which are able to adapt their encoding based on the value distribution yield better results.

6. Overall Scalability of the Retrieval System

In the previous experiments, we have seen that AFOR-3 is the most suitable compression technique for the node-based inverted index. Based on these results, we decide to combine AFOR-3 and the node-based index in a large scale experiment which simulates the conditions of a real Web Data search engine such as Sindice. We use the full Sindice data collection to create three indexes of increasing size and we generate a set of star queries of increasing complexity. We compare the query rate (queries per second) that the system can answer with respect to the size of the index and the complexity of the query.

Experimental Settings and Design. The experimental settings and design are identical as the ones found in Section 5.3.

Data Collection. We use the full Sindice data collection which is currently composed of more than 120 millions of documents among 90.000 datasets. For each dataset, we extracted the entities as pictured in Figure 3. We filtered out all the entity descriptions containing less than two facts. After filtering, there is a total of 907,542,436 entities for 4,689,599,183 RDF statements. We create three datasets: *Small* containing 226,129,319 entities for 1,240,674,545 RDF statements; *Medium* containing 447,305,647 entities for 2,535,658,099 RDF statements; and *Large* containing the complete collection of entities.

Query Benchmark Design. We generate star queries of increasing complexity, starting with 1 attribute query up to 16. Each attribute query is generated by selecting at random (following a uniform distribution) an attribute term from the high, medium or low selectivity ranges. The associated value query is generated by selecting at random (following a uniform distribution) a conjunction (2-AND or 4-AND) or a disjunction (2-OR or 4-OR). Each term of the value query is selected from the high, medium or low selectivity ranges at random (following a uniform distribution). Such a query generation scheme provides star queries of average complexity, i.e., queries composed of terms from any selectivity range.

With respect to the creation of the three selectivity ranges for the value terms, we observed the presence of a longer tail in the term frequency distribution compared to the previous experiment. Consequently, we modified the way the ranges are computed. The high range represents the first k words whose cumulative frequency is 50% of all word occurrences. The medium range accounts for the next 30%, and the low range is composed of all the remaining words.

For each type of star query, we (1) generate a set of 400 random queries, and (2) perform 100 measurements. Each measurement is made as explained in Section 5.3.2 using *warm cache*. A measurement records the query rate, i.e., the number of query the system can process per second, using a single thread. As recommended in [79], we report the harmonic mean and the standard deviation of the 100 measurements.

The design of the scalability benchmark includes three factors:

Dataset having three levels: Small, Medium and Large.
Query Size having five levels: 1, 2, 4, 8, and 16.
Term Selectivity having two levels: Low-Medium-High (LMH) and Medium-High (MH).

Each condition of the design, e.g., Small / 4 / LMH, contains 100 separate measurements. The term selectivity denotes the selectivity ranges that has been used to generate the query terms. For example, the MH selectivity level means that all the query terms have been generated from either the medium or high range.

6.1. Indexing Performance

We report that during the indexing of the data collection per batch of 100,000 entities, the commit time stayed constant, with an average commit time of 2062 milliseconds. The optimisation of the full index were performed in 119 minutes. The size of the five inverted files is 19.279 GB, with 10.912 GB for the entity file, 0.684 GB for the frequency file, 3.484 GB for the attribute file, 1.810 GB for the value file and 2.389 GB for the position file. The size of the dictionary is 8.808 GB and the size of the skip lists, i.e., the data structure for self-indexing, is 7.644 GB. The total size of the index is 35.731 GB which represents an average of 8 bytes per RDF statement.

6.2. Querying Performance

We report the results of the scalability benchmark in Table C.6. Based on these results, we derive two graphical charts in Figure 17 to better visualise the evolution of the query rate with respect to the size of the dataset and the complexity of the queries.

With respect to the size of the queries, we can observe that the query rate increases with the number of attribute value pairs until a certain point (up to 2 or 4 pairs), and then starts to decrease. The lowest query rate is obtained when the star query is composed of only one attribute query. Such a query produces a higher number of hits compared to other queries, and as a consequence the system has to read more data. For example, in Table C.6, we can see that the amount of data read is at least three times higher than in any other queries. On the other hand, the precision of the query increases with the number of attribute queries, and the chance of having a large number of hits decreases consequently. In that case, the self-indexing technique provides considerable benefits since it enables the system to avoid a large amount of unnecessary record comparisons and to answer complex queries in sub-linear time.

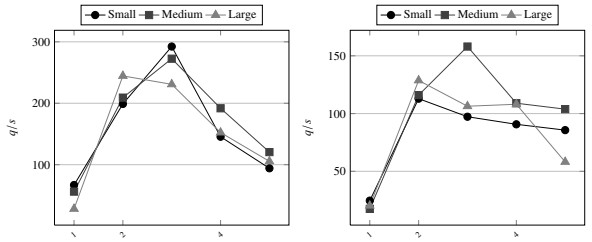
Concerning the term selectivity, we can note a drop of query rate between Figure 17a where query terms of low selectivity are employed and Figure 17b where query terms of low selectivity is not employed. In the later case, the system has to perform more record comparisons. Whenever a term with a low selectivity is used in the query, the system is able to take advantage of the self-indexing and to skip a larger number of records during query processing.

The size of the data collection has only a limited impact on the query rate. The reason is that the query processing complexity is bound by the size of the inverted lists which is itself dependent of the term distribution. Therefore, the size of the data collection has a weak influence on the size of the inverted lists, apart for very frequent terms. A term with a low or medium selectivity will have a short inverted lists even if the data collection is very large.

To conclude, the results show that the query rate of the system scales gracefully with the size of the data and the complexity of the query. A single-threaded system is able to sustain a query rate of 17 queries per second up to 292 queries per second depending of the kind of queries. At this rate, the system is able to support many requests, or users, at the same time.

7. Conclusions and Future Work

In this article, we have introduced an entity retrieval model for decentralised infrastructures providing semi-structured data such as the Semantic Web. The Entity Attribute-Value model is generic enough to be compatible with various semi-structured data models such as



(a) Query rate with LMH selectiv-(b) Query rate with MH selectivity ity

Figure 17: The evolution of the average query rate with respect to the size of the star queries over different dataset size.

RDF or Microformats. This retrieval model provide a common framework for the development of techniques which are applicable not only for Web Data search but on a wider variety of scenarios.

We have introduced a node-based indexing scheme that fulfils the requirements of our entity retrieval model. Compared to other entity retrieval systems, such a retrieval system is able to provide semi-structured search while sustaining fast query processing and efficient index maintenance. Since the performance of the retrieval system is a key issue in large web search engines, we have developed a high-performance compression technique which is particularly efficient with respect to the node-based inverted index described in this article. Finally, we have shown that the resulting retrieval system can index billions of data objects and can answer a large number of complex requests per second on hardware less powerful than the average laptop available today.

Future work will concentrate on increasing the query expressiveness of the system. We have to investigate the feasibility of path-based queries which will enable to query relations between entities. Supporting such queries while keeping a system in the same class of scalability is still an open problem.

Appendix A. Examples of Algebra Queries

Value Query.

Q1:. Find all entities with a value matching keywords *renaud* and *delbru*.

$$\begin{aligned} Q1 &= \pi_e(\sigma_{v:renaud}(\sigma_{v:delbru}(R))) \\ &= \pi_e(\sigma_{v:renaud \wedge v:delbru}(R)) \end{aligned} \quad \text{(Q1)}$$

Attribute Query.

Q2:. Find all entities with a value matching keywords *john* and *smith* associated to an attribute matching the keyword *author*.

$$\begin{aligned} Q2 &= \pi_e(\sigma_{at:author}(\sigma_{v:john \wedge v:smith}(R))) \\ &= \pi_e(\sigma_{at:author \wedge v:john \wedge v:smith}(R)) \end{aligned} \quad \text{(Q2)}$$

Star Query.

Q3:. Find all entities with a value matching keywords *john* and *smith* associated to an attribute matching the keyword *author*, and a value matching keywords *search* and *engine* associated to an attribute matching the keyword *title*.

$$\begin{aligned} R_1 &= \pi_e(\sigma_{at:author}(\sigma_{v:john \wedge v:smith}(R))) \\ R_2 &= \pi_e(\sigma_{at:title}(\sigma_{v:search \wedge v:engine}(R))) \\ Q3 &= R_1 \cap R_2 \end{aligned} \quad \text{(Q3)}$$

Dataset Query.

Q4:. Find all entities from the dataset *biblio* with a value matching keywords *john* and *smith* associated to an attribute matching the keyword *author*, and a value matching keywords *search* and *engine* associated to an attribute matching the keyword *title*.

$$\begin{aligned} R_1 &= \pi_{d,e}(\sigma_{at:author}(\sigma_{v:john \wedge v:smith}(R))) \\ R_2 &= \pi_{d,e}(\sigma_{at:title}(\sigma_{v:search \wedge v:engine}(R))) \\ Q4 &= \pi_e(\sigma_{d:biblio}(R_1 \cap R_2)) \end{aligned} \quad \text{(Q4)}$$

Q5:. Find all datasets with two entities, the first one with a value matching keywords *john* and *smith* associated to an attribute matching the keyword *name*, and the second one with a value matching keywords *search* and *engine* associated to an attribute matching the keyword *title*.

$$\begin{aligned} R_1 &= \pi_d(\sigma_{at:name}(\sigma_{v:john \wedge v:smith}(R))) \\ R_2 &= \pi_d(\sigma_{at:title}(\sigma_{v:search \wedge v:engine}(R))) \\ Q5 &= R_1 \cap R_2 \end{aligned} \quad \text{(Q5)}$$

Appendix B. Results of the Compression Benchmark

This appendix provides tables containing the results of the benchmarks that have been performed for comparing the indexing and querying performance of the node-based index with various compression algorithms. Tables C.5 have been used for generating the charts from Section 5.3.1. Tables B.4 have been used for generating the charts from Section 5.3.2.

Appendix C. Results of the Scalability Benchmark

This appendix provides the table containing the results of the scalability benchmark. Table C.6 has been used for generating the charts from Section 6.2.

Appendix D. Source Code

The source code of SIREn is publicly available at `http://siren.sindice.com/`. The source code of the benchmark platform and of the compression algorithms as well as the datasets and raw experimental results are available on request.

References

[1] R. A. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, F. Silvestri, Challenges on Distributed Web Retrieval, in: Proceedings of the 23rd International Conference on Data Engineering, IEEE, 2007, pp. 6–20. doi:10.1109/ICDE.2007.367846.

[3] S. Abiteboul, Querying Semi-Structured Data, in: Proceedings of the 6th International Conference on Database Theory, 1997, pp. 1–18.

[4] J. Pound, P. Mika, H. Zaragoza, Ad-hoc object retrieval in the web of data, in: Proceedings of the 19th international conference on World Wide Web, ACM Press, New York, New York, USA, 2010, pp. 771–780. doi:10.1145/1772690.1772769.

[5] R. Baeza-Yates, G. Navarro, Integrating contents and structure in text retrieval, SIGMOD Rec. 25 (1) (1996) 67–79. doi:http://doi.acm.org/10.1145/381854.381890.

[6] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram, XRANK: ranked keyword search over XML documents, in: Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03, ACM Press, New York, New York, USA, 2003, pp. 16–27. doi:10.1145/872757.872762.

[7] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv, XSEarch: a semantic search engine for XML, in: Proceedings of the 29th international conference on Very large data bases - VLDB '2003, VLDB Endowment, 2003, pp. 45–56.

[8] Z. Liu, J. Walker, Y. Chen, XSeek: a semantic XML search engine using keywords, in: Proceedings of the 33rd international conference on Very large data bases, 2007, pp. 1330–1333.

[9] R. Schenkel, A. Theobald, G. Weikum, Semantic Similarity Search on Semistructured Data with the XXL Search Engine, Information Retrieval 8 (4) (2005) 521–545. doi:10.1007/s10791-005-0746-3.

[10] M. Theobald, R. Schenkel, G. Weikum, An Efficient and Versatile Query Engine for TopX Search, in: Proceedings of the 31st international conference on Very Large Data Bases, 2005, pp. 625–636.

[11] N. Walsh, M. Fernández, A. Malhotra, M. Nagy, J. Marsh, XQuery 1.0 and XPath 2.0 data model (XDM), W3C recommendation, W3C (January 2007).

[12] Q. Li, B. Moon, Indexing and Querying XML Data for Regular Path Expressions, in: Proceedings of the 27th International Conference on Very Large Data Bases, 2001, pp. 361–370.

[13] H. He, H. Wang, J. Yang, P. S. Yu, Compact reachability labeling for graph-structured data, in: Proceedings of the 14th ACM international conference on Information and knowledge management - CIKM '05, ACM Press, New York, New York, USA, 2005, pp. 594–601. doi:10.1145/1099554.1099708.

[14] W. Haixun, H. Hao, Y. Jun, P. Yu, J. Yu, Dual Labeling: Answering Graph Reachability Queries in Constant Time, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), IEEE, 2006, pp. 75–75. doi:10.1109/ICDE.2006.53.

[15] R. Goldman, J. Widom, DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, in: Proceedings of the 23rd International Conference on Very Large Data Bases, 1997, pp. 436–445.

[16] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, M. Shadmon, A Fast Index for Semistructured Data, in: Proceedings of the 27th International Conference on Very Large Data Bases, 2001, pp. 341–350.

[17] W. Wang, H. Jiang, H. Wang, X. Lin, H. Lu, J. Li, Efficient processing of XML path queries using the disk-based F&B Index, in: Proceedings of the 31st international conference on Very Large Data Bases, 2005, pp. 145–156.

[18] H. Wang, S. Park, W. Fan, P. S. Yu, ViST: a dynamic index method for querying XML data by tree structures, in: Proceedings of the 2003 ACM SIGMOD international conference on on Management of data, ACM Press, New York, New York, USA, 2003, pp. 110–121. doi:10.1145/872757.872774.

[19] P. Roa, B. Moon, PRIX: indexing and querying XML using prufer sequences, in: Proceedings of the 20th International Conference on Data Engineering, IEEE Computer Society, 2004, pp. 288–299. doi:10.1109/ICDE.2004.1320005.

[20] X. Meng, Y. Jiang, Y. Chen, H. Wang, XSeq: an indexing infrastructure for tree pattern queries, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04, ACM Press, New York, New York, USA, 2004, pp. 941–942. doi:10.1145/1007568.1007709.

[21] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and searching XML data via two zips, in: Proceedings of the 15th international conference on World Wide Web - WWW '06, ACM Press, New York, New York, USA, 2006, pp. 751–760. doi:10.1145/1135777.1135891.

[22] N. Grimsmo, Faster Path Indexes for Search in XML Data, in: Proceedings of the nineteenth conference on Australasian database, 2008, pp. 127–135.

[23] H. Su-Cheng, L. Chien-Sing, Node Labeling Schemes in XML Query Optimization: A Survey and Trends, IETE Technical Review 26 (2) (2009) 88. doi:10.4103/0256-4602.49086.

[24] A. Halevy, M. Franklin, D. Maier, Principles of dataspace systems, in: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, Chicago, IL, USA, 2006.

[25] X. Dong, A. Halevy, Indexing dataspaces, Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07 (2007) 43doi:10.1145/1247480.1247487.

[26] D. Beckett, J. Grant, Semantic Web Scalability and Storage: Mapping Semantic Web Data with RDBMSes, SWAD-Europe deliverable, W3C (January 2003).

[27] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, in: Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference, Vol. 4825 of Lecture Notes in Computer Science, Springer Verlag, 2007, pp. 211–224.

[28] L. Baolin, H. Bo, HPRD: A High Performance RDF Database, in: Proceedings of Network and Parallel Computing, IFIP International Conference, Vol. 4672 of Lecture Notes in Computer Science, Springer, 2007, pp. 364–374.

[29] C. Weiss, P. Karras, A. Bernstein, Hexastore - sextuple indexing for semantic web data management, Proceedings of the VLDB Endowment 1 (1) (2008) 1008–1019. doi:10.1145/1453856.1453965.

[30] T. Neumann, G. Weikum, RDF-3X - a RISC-style Engine for RDF, Proceedings of the VLDB Endowment 1 (1) (2008) 647–659. doi:10.1145/1453856.1453927.

[31] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment, 2007, pp. 411–422.

[32] R. Guha, R. McCool, E. Miller, Semantic search, in: Proceedings of the 12th international conference on World Wide Web, 2003, pp. 700–709.

[33] L. Ding, R. Pan, T. Finin, A. Joshi, Y. Peng, P. Kolari, Finding and Ranking Knowledge on the Semantic Web, in: Proceedings of the 4th International Semantic Web Conference, 2005, pp. 156–170.

[34] A. Harth, A. Hogan, J. Umbrich, S. Decker, SWSE: Objects before documents!, in: Proceedings of the Billion Triple Semantic Web Challenge, 7th International Semantic Web Conference, 2008.

[35] G. Cheng, W. Ge, Y. Qu, Falcons: searching and browsing entities on the semantic web, in: Proceeding of the 17th international conference on World Wide Web, ACM, 2008, pp. 1101–1102.

[36] R. Bhagdev, S. Chapman, F. Ciravegna, V. Lanfranchi, D. Petrelli, Hybrid Search: Effectively Combining Keywords and Semantic Searches, in: Proceedings of the 5th European semantic web conference on The semantic web: research and applications, Springer-Verlag, 2008, pp. 554–568.

[37] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, Y. Pan, 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.

[38] S. Agrawal, S. Chaudhuri, G. Das, DBXplorer: a system for keyword-based search over relational databases, in: Proceedings of the 18th International Conference on Data Engineering, IEEE Comput. Soc, San Jose, California, 2002, pp. 5–16. doi:10.1109/ICDE.2002.994693.

[39] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan, Keyword searching and browsing in databases using BANKS, in: Proceedings of the 18th International Conference on Data Engineering, IEEE Comput. Soc, 2002, pp. 431–440. doi:10.1109/ICDE.2002.994756.

[40] V. Hristidis, Y. Papakonstantinou, Discover: keyword search in relational databases, in: Proceedings of the 28th international conference on Very Large Data Bases, 2002, pp. 670–681.

[41] F. Liu, C. Yu, W. Meng, A. Chowdhury, Effective keyword search in relational databases, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06, ACM Press, Chicago, IL, USA, 2006, p. 563. doi:10.1145/1142473.1142536.

[42] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener, The Lorel query language for semistructured data, International Journal on Digital Libraries 1 (1996) 68–88. doi:10.1007/s007990050005.

[43] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, H. Karambelkar, Bidirectional expansion for keyword search on graph databases, in: Proceedings of the 31st international conference on Very large data bases, Trondheim, Norway, 2005, pp. 505–516.

[44] G. Li, B. C. Ooi, J. Feng, J. Wang, L. Zhou, EASE: an effective 3-in-1 keyword search method for unstructured, semistructured and structured data, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08, Vancouver, Canada, 2008, pp. 903–914. doi:10.1145/1376616.1376706.

[45] G. Kasneci, F. M. Suchanek, G. Ifrim, S. Elbassuoni, M. Ramanath, G. Weikum, NAGA: harvesting, searching and ranking knowledge, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08, Vancouver, Canada, 2008, pp. 1285–1288. doi:10.1145/1376616.1376756.

[46] F. Mandreoli, R. Martoglia, G. Villani, W. Penzo, Flexible query answering on graph-modeled data, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, ACM, Saint Petersburg, Russia, 2009, pp. 216—-227. doi:10.1145/1516360.1516386.

[47] G. H. L. Fletcher, J. V. D. Bussche, D. V. Gucht, S. Vansummeren, Towards a theory of search queries, in: Proceedings of the 12th International Conference on Database Theory, 2009, pp. 201—-211.

[48] M. Catasta, R. Delbru, N. Toupikov, G. Tummarello, Managing Terabytes of Web Semantics Data, Springer, 2010, p. 93. doi:10.1007/978-3-642-04329-1_6.

[49] R. Delbru, N. Toupikov, M. Catasta, G. Tummarello, S. Decker, Hierarchical Link Analysis for Ranking Web Data, in: Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010), Springer, 2010, pp. 240–256. doi:10.1007/978-3-642-13489-0_17.

[50] T. Berners-Lee, R. T. Fielding, H. F. Nielsen, Hypertext Transfer Protocol – HTTP/1.0, RFC 1945, W3C (May 1996).

[51] D. Beckett, RDF/XML Syntax Specification (Revised), W3C recommendation, W3C (February 2004).

[52] A. Harth, S. Kinsella, S. Decker, Using Naming Authority to Rank Data and Ontologies for Web Search , in: The Semantic Web - ISWC 2009, Springer Berlin Heidelberg, 2009, pp. 277 – 292. doi:10.1007/978-3-642-04930-9.

[53] P. M. Nadkarni, L. Marenco, R. Chen, E. Skoufos, G. Shepherd, P. Miller, Organization of heterogeneous scientific data using the EAV/CR representation., Journal of the American Medical Informatics Association : JAMIA 6 (6) (1999) 478–493. doi:10.1136/jamia.1999.0060478.

[54] D. Maier, Theory of Relational Databases, Computer Science Press, 1983.

[55] R. Delbru, N. Toupikov, M. Catasta, G. Tummarello, S. Decker, A Node Indexing Scheme for Web Entity Retrieval, in: Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010), Springer, 2010, pp. 225–239. doi:10.1007/978-3-642-13489-0_16.

[56] K. Beyer, S. D. Viglas, I. Tatarinov, J. Shanmugasundaram, E. Shekita, C. Zhang, Storing and querying ordered xml using a relational database system, in: SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of Data, ACM, New York, NY, USA, 2002, pp. 204–215. doi:10.1145/564691.564715.

[57] J. Zobel, A. Moffat, Inverted files for text search engines, ACM Computer Surveys 38 (2) (2006) 6. doi:10.1145/1132956.1132959.

[58] C. D. Manning, P. Raghavan, H. Schtze, Introduction to Information Retrieval, Cambridge University Press, New York, NY, USA, 2008.

[59] A. Moffat, J. Zobel, Self-indexing inverted files for fast text retrieval, ACM Trans. Inf. Syst. 14 (4) (1996) 349–379. doi:10.1145/237496.237497.

[60] G. Graefe, Query evaluation techniques for large databases, ACM Computing Surveys 25 (2) (1993) 73. doi:10.1145/152610.152611.

[61] G. Graefe, B-tree indexes for high update rates, ACM SIGMOD

Record 35 (1) (2006) 39. doi:10.1145/1121995.1122002.

[62] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, R. Agarwal, Dynamic maintenance of web indexes using landmarks, in: Proceedings of the 12th international conference on World Wide Web, 2003, p. 102. doi:10.1145/775152.775167.

[63] S. Büttcher, C. L. A. Clarke, Index compression is good, especially for random access, in: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management - CIKM '07, ACM Press, New York, New York, USA, 2007, pp. 761–770. doi:10.1145/1321440.1321546.

[64] J. Goldstein, R. Ramakrishnan, U. Shaft, Compressing relations and indexes, in: Proceedings of the 14th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1998, pp. 370–379. doi:10.1109/ICDE.1998.655800.

[65] V. N. Anh, A. Moffat, Inverted Index Compression Using Word-Aligned Binary Codes, Information Retrieval 8 (1) (2005) 151–166. doi:10.1023/B:INRT.0000048490.99518.5c.

[66] M. Zukowski, S. Heman, N. Nes, P. Boncz, Super-Scalar RAM-CPU Cache Compression, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 59–59. doi:10.1109/ICDE.2006.150.

[67] V. N. Anh, A. Moffat, Index compression using 64-bit words, Software: Practice and Experience 40 (2) (2010) 131–147. doi:10.1002/spe.948.

[68] P. Boldi, S. Vigna, Compressed Perfect Embedded Skip Lists for Quick Inverted-Index Lookups, in: M. Consens, G. Navarro (Eds.), Proceedings of the 12th International Conference on String Processing and Information Retrieval, Vol. 3772 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 25–28. doi:10.1007/11575832.

[69] J. Zhang, X. Long, T. Suel, Performance of compressed inverted list caching in search engines, in: Proceeding of the 17th international conference on World Wide Web - WWW '08, ACM Press, New York, New York, USA, 2008, pp. 387–396. doi:10.1145/1367497.1367550.

[70] H. Yan, S. Ding, T. Suel, Inverted index compression and query processing with optimized document ordering, in: Proceedings of the 18th international conference on World Wide Web - WWW '09, ACM Press, New York, New York, USA, 2009, pp. 401–410. doi:10.1145/1526709.1526764.

[71] V. N. Anh, A. Moffat, Structured Index Organizations for High-Throughput Text Querying, in: Proceedings of the 13th International Conference of String Processing and Information Retrieval, Springer, 2006, pp. 304–315. doi:10.1007/11880561_25.

[72] R. Sacks-davis, T. Dao, J. A. Thom, J. Zobel, Indexing documents for queries on structure, content and attributes, in: Proceedings of International Symposium on Digital Media Information Base, World Scientific, 1997, pp. 236–245.

[73] A. Moffat, L. Stuiver, Binary Interpolative Coding for Effective Index Compression, Information Retrieval 3 (1) (2000) 25–47. doi:10.1023/A:1013002601898.

[74] R. Rice, J. Plaunt, Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data, IEEE Transactions on Communications 19 (6) (1971) 889–897. doi:10.1109/TCOM.1971.1090789.
URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1090789`

[75] R. Venturini, F. Silvestri, Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming, in: Proceedings of the nineteenth ACM conference on Conference on information and knowledge management- CIKM'10, 2010.

[76] B. Boyer, Robust Java benchmarking (2008).
URL `http://www.ibm.com/developerworks/java/library/j-benchmark1.html`

[77] S. Büttcher, C. L. A. Clarke, Hybrid index maintenance for contiguous inverted lists, Information Retrieval 11 (3) (2008) 175–207. doi:10.1007/s10791-007-9042-8.

[78] V. Ercegovac, D. J. DeWitt, R. Ramakrishnan, The TEXTURE benchmark: measuring performance of text queries on a relational DBMS, in: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, 2005, pp. 313–324.

[79] D. J. Lilja, Measuring Computer Performance: A Practitioner's Guide, Cambridge University Press, 2000. doi:10.1017/CBO9780511612398.

	2 - AND				2 - OR				4 - AND				4 - OR				2 - Phrase				3 - Phrase			
Method	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB
DBpedia																								
AFOR-1	32.6	±0.1	1.3	1.8	42.9	±0.1	1.2	2.0	63.2	±0.2	2.4	3.6	88.4	±0.2	2.5	4.0	218.4	$^{+2.1}_{-1.5}$	13.7	14.2	508.3	±0.8	4.4	36.6
AFOR-2	37.7	±0.1	1.2	1.7	47.8	±0.1	1.7	1.9	74.1	±0.4	3.0	3.3	98.6	±0.2	2.5	3.7	253.2	$^{+1.4}_{-1.9}$	12.9	13.2	569.3	±0.8	4.6	33.8
AFOR-3	44.2	±0.1	1.2	1.7	52.3	$^{+0.6}_{-0.8}$	11.2	1.8	86.0	±0.5	3.7	3.3	110.1	$^{+0.3}_{-0.6}$	5.0	3.6	256.7	$^{+1.6}_{-2.0}$	13.9	13.1	593.9	$^{+5.9}_{-5.6}$	32.2	33.6
FOR	31.5	±0.1	1.4	2.3	46.8	$^{+0.4}_{-0.6}$	10.6	2.5	61.9	±0.3	2.9	4.5	86.3	±0.2	2.5	5.1	220.5	$^{+2.3}_{-1.3}$	13.2	18.9	531.1	$^{+7.0}_{-5.8}$	35.3	48.5
PFOR	44.2	$^{+0.8}_{-0.7}$	17.1	2.3	52.2	±0.1	1.3	2.5	83.1	$^{+0.3}_{-0.2}$	2.7	4.5	106.8	$^{+0.2}_{-0.3}$	2.6	5.0	225.1	±0.3	2.7	16.5	521.1	$^{+0.9}_{-0.8}$	4.5	41.9
Rice	75.4	±0.1	1.6	1.7	98.8	$^{+0.5}_{-0.7}$	8.9	1.8	148.0	±0.4	3.1	3.3	190.5	±0.5	3.7	3.7	604.8	$^{+0.8}_{-0.9}$	4.9	12.1	1573.0	±1.7	6.4	29.9
S-64	42.4	$^{+0.3}_{-0.1}$	3.0	1.9	57.8	±0.1	1.6	2.0	83.3	±0.2	2.4	3.6	117.8	±0.2	2.4	4.0	291.0	$^{+2.0}_{-1.8}$	15.3	13.7	668.8	$^{+1.0}_{-1.1}$	6.0	35.0
VByte	45.8	$^{+0.7}_{-0.8}$	17.8	2.7	57.2	±0.1	1.5	2.9	81.0	±0.3	2.9	5.2	116.7	±0.2	2.3	5.9	330.5	$^{+1.3}_{-2.3}$	13.0	21.9	723.9	$^{+1.0}_{-1.1}$	5.8	57.8
Geonames																								
AFOR-1	29.3	±0.1	1.5	1.4	30.7	±0.4	9.2	1.4	62.7	$^{+0.7}_{-0.9}$	8.8	2.9	59.0	$^{+0.3}_{-0.2}$	2.8	2.9	35.3	±0.1	1.8	1.7	60.6	±0.1	2.0	3.1
AFOR-2	36.6	$^{+0.1}_{-0.3}$	4.3	1.4	33.7	±0.1	1.6	1.4	69.3	±0.8	8.5	2.9	72.0	$^{+0.7}_{-0.9}$	8.6	2.9	40.4	$^{+0.2}_{-0.1}$	2.4	1.6	68.1	±0.2	2.5	2.8
AFOR-3	32.6	±0.1	1.7	1.3	32.8	±0.1	1.4	1.3	65.5	±0.3	3.2	2.7	66.0	$^{+0.2}_{-0.3}$	2.7	2.7	40.1	±0.1	1.8	1.5	79.5	±0.2	2.5	2.7
FOR	30.4	±0.4	8.3	1.5	31.7	±0.1	1.5	1.5	63.4	±0.8	4.6	3.0	63.4	±0.3	2.9	3.0	35.8	$^{+0.9}_{-0.7}$	12.4	2.2	58.9	$^{+0.7}_{-0.5}$	4.5	4.1
PFOR	37.8	±0.1	2.1	1.5	38.1	±0.1	1.9	1.5	78.2	±0.9	9.8	3.0	77.1	$^{+0.5}_{-0.4}$	4.7	3.0	45.7	$^{+0.8}_{-0.5}$	14.1	2.2	87.6	$^{+0.5}_{-1.2}$	10.5	4.0
Rice	69.0	±0.1	2.1	1.5	69.4	±0.2	2.9	1.5	141.0	±0.8	6.5	3.0	139.0	±0.5	3.9	3.0	89.4	$^{+1.1}_{-1.0}$	11.9	1.8	134.3	±0.4	2.9	3.2
S-64	41.0	±0.1	2.3	1.8	42.5	$^{+0.3}_{-0.4}$	8.0	1.8	82.8	$^{+0.4}_{-0.3}$	3.8	3.6	80.7	±0.3	2.8	3.6	53.9	±0.1	2.2	1.8	75.7	±0.3	2.8	3.1
VByte	40.2	±0.1	1.4	2.9	39.8	±0.1	1.3	2.9	85.7	$^{+0.6}_{-0.9}$	8.0	5.8	80.7	±0.2	2.1	5.8	46.7	±0.1	1.3	3.2	77.8	±0.1	1.9	5.7
Sindice																								
AFOR-1	31.4	±0.1	1.3	1.8	40.6	$^{+0.0}_{-0.1}$	1.1	1.9	76.7	±0.2	2.0	3.5	83.6	$^{+1.7}_{-1.8}$	19.9	3.7	300.3	$^{+0.3}_{-0.4}$	2.9	19.1	1377.0	$^{+1.3}_{-1.5}$	5.7	78.8
AFOR-2	36.8	±0.1	1.4	1.6	51.9	$^{+0.9}_{-1.0}$	14.6	1.7	73.5	±0.2	2.3	3.2	99.3	$^{+1.0}_{-1.5}$	13.3	3.4	329.9	±0.4	3.2	17.5	1394.0	±1.5	5.9	72.4
AFOR-3	36.3	±0.1	1.3	1.6	45.6	±0.1	1.2	1.7	72.0	±0.4	3.0	3.1	85.6	±0.2	2.3	3.2	325.0	$^{+0.7}_{-0.8}$	4.1	16.9	1377.0	$^{+1.5}_{-1.7}$	6.1	70.4
FOR	35.9	$^{+0.7}_{-0.9}$	17.9	2.3	37.3	±0.1	1.1	2.4	60.1	±0.3	2.3	4.5	70.5	$^{+0.1}_{-0.2}$	2.0	4.7	323.6	$^{+3.9}_{-3.8}$	30.3	28.3	1382.0	$^{+2.0}_{-1.8}$	7.8	116.3
PFOR	40.5	±0.1	1.7	2.3	49.8	±0.1	2.1	2.4	81.4	$^{+1.3}_{-0.7}$	10.0	4.5	94.1	$^{+0.3}_{-0.2}$	2.4	4.7	316.6	±0.4	3.1	25.5	1282.0	$^{+1.5}_{-1.7}$	6.4	103.0
Rice	68.5	±0.2	2.0	1.8	82.4	±0.1	1.5	1.9	151.0	±0.5	3.7	3.6	155.8	±0.4	2.9	3.7	848.3	$^{+2.3}_{-3.1}$	14.9	18.6	3348.0	$^{+1.8}_{-1.7}$	6.7	74.0
S-64	40.9	±0.1	1.4	1.8	52.5	±0.1	1.9	1.9	81.1	$^{+0.3}_{-0.2}$	2.7	3.6	97.9	±0.2	2.3	3.8	408.6	$^{+2.5}_{-1.9}$	17.1	18.0	1700.0	$^{+3.2}_{-3.0}$	12.2	74.5
VByte	40.3	$^{+0.1}_{-0.0}$	1.1	2.8	61.1	±0.1	1.7	3.0	79.5	±0.2	2.2	5.5	111.5	$^{+1.0}_{-1.7}$	14.6	5.8	462.3	$^{+3.8}_{-4.3}$	31.7	31.5	1843.0	$^{+1.8}_{-1.5}$	6.7	133.7

(a) Value Query

	2 - AND				2 - OR				4 - AND				4 - OR				2 - Phrase				3 - Phrase			
Method	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB
DBpedia																								
AFOR-1	47.1	±0.1	1.6	2.4	134.6	±0.2	2.2	7.3	87.4	$^{+1.3}_{-1.6}$	16.5	4.1	200.1	$^{+0.5}_{-0.4}$	3.6	10.7	244.0	$^{+0.3}_{-0.4}$	2.8	15.3	564.1	$^{+5.3}_{-5.9}$	31.2	37.6
AFOR-2	64.0	±0.1	2.1	2.2	132.5	±0.3	2.7	6.8	103.0	$^{+1.2}_{-1.6}$	15.3	3.8	220.0	$^{+1.0}_{-1.8}$	10.3	9.9	282.3	$^{+1.8}_{-2.6}$	17.0	14.2	594.4	$^{+3.6}_{-2.2}$	15.2	34.7
AFOR-3	54.5	±0.1	2.2	2.1	136.0	±0.2	2.1	5.9	104.1	$^{+0.6}_{-1.1}$	8.5	3.7	190.3	$^{+0.4}_{-0.5}$	3.4	8.7	264.0	±0.4	3.2	13.9	600.4	±0.8	4.3	34.4
FOR	54.5	±0.1	2.2	2.1	136.0	±0.2	2.1	5.9	104.1	$^{+0.6}_{-1.1}$	8.5	3.7	190.3	$^{+0.4}_{-0.5}$	3.4	8.7	264.0	±0.4	3.2	13.9	600.4	±0.8	4.3	34.4
PFOR	61.3	$^{+0.5}_{-0.3}$	4.7	3.1	146.1	±0.2	2.4	8.7	117.1	±0.5	4.2	5.3	199.3	±0.5	3.9	12.7	249.3	$^{+0.5}_{-0.4}$	3.5	18.0	578.2	$^{+5.6}_{-6.1}$	32.6	43.3
Rice	107.0	±0.2	2.4	2.3	312.2	±0.4	3.2	6.8	192.8	$^{+0.5}_{-0.4}$	3.2	3.9	475.5	$^{+2.0}_{-2.4}$	12.2	9.8	677.0	$^{+1.0}_{-0.9}$	5.2	13.2	1625.0	$^{+2.0}_{-1.8}$	7.6	30.9
S-64	64.0	$^{+1.2}_{-1.0}$	12.1	2.4	144.5	$^{+0.4}_{-0.5}$	4.9	6.9	103.7	$^{+0.4}_{-0.3}$	3.9	4.1	215.0	$^{+0.6}_{-0.5}$	4.6	10.1	316.9	±0.5	3.7	14.7	706.3	$^{+1.1}_{-0.9}$	5.4	35.9
VByte	59.0	±0.1	1.9	3.8	165.6	±0.2	2.3	14.8	110.8	$^{+1.3}_{-1.8}$	16.6	6.3	264.8	$^{+2.3}_{-3.2}$	21.0	20.8	339.9	$^{+0.3}_{-0.4}$	2.9	24.1	767.3	$^{+7.3}_{-6.1}$	37.1	59.8
Geonames																								
AFOR-1	42.9	±0.1	2.1	1.7	84.0	$^{+0.3}_{-0.2}$	2.7	2.4	71.9	±0.2	2.5	3.2	117.9	±0.3	3.5	4.4	64.2	±0.1	2.0	2.1	78.8	±0.2	2.5	3.4
AFOR-2	55.6	$^{+1.1}_{-1.3}$	18.7	1.7	91.2	$^{+0.1}_{-0.2}$	1.9	2.3	85.9	$^{+1.9}_{-1.5}$	19.1	3.1	129.8	±0.5	3.6	4.3	59.8	$^{+0.3}_{-0.2}$	2.7	2.0	90.1	±0.3	3.3	3.2
AFOR-3	50.5	$^{+1.3}_{-1.1}$	19.6	1.5	70.2	±0.1	2.0	1.9	89.8	$^{+2.0}_{-2.2}$	23.4	2.9	137.2	$^{+2.8}_{-3.2}$	23.8	3.5	69.9	$^{+0.9}_{-1.3}$	11.6	1.7	87.5	$^{+0.5}_{-0.4}$	3.5	2.9
FOR	41.6	±0.2	2.6	1.9	80.5	±0.1	2.2	2.6	68.8	±0.3	2.9	3.4	111.3	±0.5	3.7	4.6	51.9	±0.2	2.6	2.7	77.4	±0.5	3.9	4.7
PFOR	56.3	±0.3	3.2	2.0	81.0	±0.3	2.9	2.7	94.1	$^{+0.2}_{-0.3}$	2.8	3.5	137.5	±0.6	4.6	4.7	67.4	±0.2	3.1	2.8	98.2	±0.3	3.5	4.5
Rice	97.5	$^{+0.2}_{-0.3}$	2.7	1.9	158.1	$^{+1.1}_{-0.5}$	5.7	2.5	165.2	±0.5	4.2	3.4	272.8	$^{+0.4}_{-0.3}$	2.8	4.6	120.8	±0.2	2.6	2.2	173.5	±0.4	3.2	3.6
S-64	60.6	$^{+1.4}_{-1.3}$	21.1	2.1	83.7	$^{+0.3}_{-0.2}$	3.0	2.6	96.0	$^{+0.6}_{-0.4}$	3.8	3.8	168.9	±0.5	4.0	4.9	75.4	$^{+1.5}_{-1.6}$	17.7	2.2	99.3	±0.3	3.4	3.5
VByte	57.9	$^{+1.3}_{-1.0}$	17.6	3.9	91.9	±0.2	2.5	6.8	95.2	±0.3	3.4	6.8	159.2	±0.5	3.5	12.5	82.7	$^{+0.2}_{-0.1}$	2.2	4.5	116.1	$^{+2.0}_{-2.5}$	24.5	6.9
Sindice																								
AFOR-1	55.5	±0.1	1.9	2.1	192.9	±0.2	2.4	8.2	77.8	±0.2	2.4	3.9	311.1	±0.4	3.1	14.4	310.3	$^{+0.8}_{-0.7}$	4.0	19.0	1297.0	$^{+1.7}_{-1.5}$	6.2	78.0
AFOR-2	53.3	±0.1	1.6	1.9	229.2	±0.4	3.0	7.5	105.1	$^{+0.8}_{-1.4}$	11.3	3.5	330.7	$^{+4.8}_{-3.5}$	32.3	13.2	341.0	$^{+0.7}_{-0.8}$	4.0	17.4	1484.0	$^{+1.3}_{-1.5}$	5.6	71.7
AFOR-3	52.1	±0.1	1.9	1.8	180.2	$^{+0.2}_{-0.3}$	2.5	5.5	88.7	±0.2	2.5	3.3	291.2	±0.4	3.2	10.0	334.8	±0.4	2.9	16.6	1413.0	$^{+1.7}_{-1.3}$	5.9	69.6
FOR	46.4	$^{+0.7}_{-0.4}$	8.0	3.0	197.0	$^{+0.5}_{-0.4}$	3.2	15.3	76.2	$^{+0.2}_{-0.3}$	2.6	5.3	314.3	$^{+0.5}_{-0.4}$	3.3	25.4	319.1	±0.8	4.2	29.1	1304.0	$^{+2.0}_{-1.5}$	7.1	115.9
PFOR	67.4	$^{+1.2}_{-1.4}$	14.6	2.8	193.9	±0.4	2.9	9.2	100.5	$^{+0.4}_{-0.5}$	3.2	5.0	316.1	±0.4	3.2	17.0	358.7	±0.4	3.1	25.5	1348.0	±1.8	7.4	102.3
Rice	100.4	±0.2	2.3	2.4	481.3	$^{+0.8}_{-0.6}$	3.8	10.9	170.5	$^{+0.5}_{-0.4}$	3.4	4.1	797.8	$^{+6.3}_{-4.8}$	30.6	18.5	825.6	$^{+1.0}_{-0.9}$	5.3	19.2	3808.0	±1.8	7.6	73.9
S-64	58.8	±0.2	2.4	2.1	213.2	$^{+2.5}_{-1.3}$	13.8	7.5	99.9	±0.2	2.3	3.9	342.7	$^{+0.7}_{-0.8}$	4.0	13.3	416.9	$^{+1.6}_{-2.6}$	16.0	17.8	1724.0	$^{+2.0}_{-1.8}$	7.8	73.7
VByte	58.8	$^{+1.1}_{-0.6}$	12.8	3.8	311.3	$^{+0.3}_{-0.4}$	3.0	25.9	97.8	±0.2	2.2	6.5	478.1	$^{+9.4}_{-9.6}$	53.0	42.5	438.4	$^{+0.6}_{-0.4}$	4.1	32.7	1916.0	$^{+1.5}_{-1.7}$	6.3	133.1

(b) Attribute Query

Table B.4: Query time execution for per query type, algorithm and dataset. We report for each query type the arithmetic mean (μ in millisecond), the confidence interval with 95% confidence level for the mean (c in millisecond), the standard deviation (σ in millisecond) and the total amount of data read during query processing (*MB* in megabyte).

	Time (s)		Sizes (GB)					
Method	Total	Opt	Ent	Frq	Att	Val	Pos	Total
AFOR-1	536	139	0.246	0.043	0.141	0.065	0.180	0.816
AFOR-2	540	142	0.229	0.039	0.132	0.059	0.167	0.758
AFOR-3	562	158	0.229	0.031	0.131	0.054	0.159	0.736
FOR	594	158	0.315	0.061	0.170	0.117	0.216	1.049
PFOR	583	167	0.317	0.044	0.155	0.070	0.205	0.946
Rice	612	239	0.240	0.029	0.115	0.057	0.152	0.708
S-64	558	162	0.249	0.041	0.133	0.062	0.171	0.791
VByte	577	164	0.264	0.162	0.222	0.222	0.245	1.335

(a) DBpedia

	Time (s)		Sizes (GB)					
Method	Total	Opt	Ent	Frq	Att	Val	Pos	Total
AFOR-1	729	114	0.129	0.023	0.058	0.025	0.025	0.318
AFOR-2	732	107	0.123	0.023	0.057	0.024	0.024	0.307
AFOR-3	724	103	0.114	0.006	0.056	0.016	0.008	0.256
FOR	748	102	0.150	0.021	0.065	0.025	0.023	0.349
PFOR	741	134	0.154	0.019	0.057	0.022	0.023	0.332
Rice	787	183	0.133	0.019	0.063	0.029	0.021	0.327
S-64	740	123	0.147	0.021	0.058	0.023	0.023	0.329
VByte	737	141	0.216	0.142	0.143	0.143	0.143	0.929

(b) Geonames

	Time (s)		Sizes (GB)					
Method	Total	Opt	Ent	Frq	Att	Val	Pos	Total
AFOR-1	13734	1816	2.578	0.395	0.942	0.665	1.014	6.537
AFOR-2	13975	1900	2.361	0.380	0.908	0.619	0.906	6.082
AFOR-3	13847	1656	2.297	0.176	0.876	0.530	0.722	5.475
FOR	14978	1749	3.506	0.506	1.121	0.916	1.440	8.611
PFOR	14839	2396	3.221	0.374	1.153	0.795	1.227	7.924
Rice	15571	3281	2.721	0.314	0.958	0.714	0.941	6.605
S-64	14107	2163	2.581	0.370	0.917	0.621	0.908	6.313
VByte	13223	3018	3.287	2.106	2.411	2.430	2.488	15.132

(c) Sindice

Table C.5: Total indexing time, optimise time and index size.

	1				2				4				8				16			
Selectivity	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB	μ	c	σ	MB
Small																				
LMH	66.9	± 0.03	0.2	165.7	198.9	$^{+1.42}_{-1.14}$	6.8	55.8	292.3	$^{+0.51}_{-1.26}$	3.6	31.6	145.5	$^{+1.3}_{-1.12}$	6.4	50.2	94.1	$^{+0.87}_{-0.82}$	4.4	65.7
MH	24.6	$^{+0.16}_{-0.15}$	0.8	424.8	112.9	$^{+0.82}_{-0.76}$	4.1	90.3	97.3	$^{+0.08}_{-0.18}$	0.6	121.6	90.7	$^{+0.65}_{-0.7}$	3.4	110.7	85.7	$^{+0.71}_{-0.73}$	3.6	91.3
Medium																				
LMH	56.2	$^{+0.04}_{-0.05}$	0.2	188.0	209.2	$^{+1.53}_{-1.52}$	7.8	52.5	272.6	$^{+1.63}_{-0.81}$	6.2	33.2	191.8	$^{+2.16}_{-2.0}$	10.7	18.7	120.5	$^{+0.12}_{-0.17}$	0.7	40.0
MH	17.5	$^{+0.03}_{-0.02}$	0.1	301.8	116.0	± 0.66	3.4	80.8	158.0	$^{+1.26}_{-1.36}$	6.5	72.6	109.0	$^{+0.98}_{-0.87}$	4.8	74.1	103.8	± 1.13	5.7	70.8
Large																				
LMH	28.1	± 0.02	0.1	377.4	244.5	$^{+0.21}_{-0.24}$	1.1	51.2	230.8	$^{+0.21}_{-0.24}$	1.2	41.6	152.3	$^{+1.64}_{-1.54}$	8.2	50.2	105.4	$^{+0.74}_{-0.97}$	4.0	58.4
MH	20.3	$^{+0.02}_{-0.04}$	0.1	543.3	128.7	± 0.1	0.5	100.1	106.4	$^{+0.1}_{-0.18}$	0.7	103.7	108.0	$^{+0.38}_{-0.64}$	2.3	95.2	58.2	$^{+0.13}_{-0.17}$	0.7	96.7

Table C.6: Query rate per dataset, term selectivity and query size. We report for each query size the arithmetic mean (μ in queries per second), the confidence interval with 95% confidence level for the mean (c in queries per second), the standard deviation (σ in queries per second) and the total amount of data read during query processing (*MB* in megabyte).