DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

DERI Galway
IDA Business Park
Lower Dangan
Galway, Ireland
http://www.deri.ie/

Compression Techniques for Inverted Lists in Web Search Engines.

Inverted Lists Structure for Semi-Structured Web Data Search.

Campinas Stéthane

End of Study Internship

Under the supervision of

Dr. Renaud Delbru and Dr. Giovanni Tummarello
September 2018

EPITA
14-16 rue Voltaire
94270 Kremlin Bicêtre
FRANCE
http://www.epita.fr/

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

DERI - EPITA — END OF STUDY REPORT

COGNITIVE SCIENCE AND ARTIFICIAL INTELLIGENCE - SEPTEMBER 2018

Abstract.
The Semantic Web is a set of layers that wraps the current web and adds useful meta-information describing web documents' content. This semantic information has a structured format, allowing web documents to be exploitable by humans and machines alike. In the latter case, machines permit a more comprehensive use of the colossal amount of knowledge in the Web. The number of such documents can already be counted in millions and is increasing rapidly. This calls for large scale systems that are able to search through this data and to retrieve relevant information. This report highlights that the time spent on IO operations for such systems is wasted, and thus reviews two approaches to answer this problem. We propose a compression method that increases both the update and the query throughput of the system. We introduce also a model that skips over data, avoiding to read unnecessary data.

[1]

[1] **Acknowledgements**:
The author wishes to thank his supervisors, Giovanni Tummarello and Renaud Delbru, for making this internship possible. The author wishes to thank also Renaud Delbru for providing a constant help and giving excellent instructions as well as an interesting view of the research world. The author also wants to thanks the EPITA teacher for the preparation during the the school year.
The author wishes further to thank his family and Katarzyna Osuchowska for their continuous support.

Contents

Part I

Prelude

Chapter 1

Introduction

The Web as we know it is mostly a huge collection of documents, e.g. texts, pictures, . . . , connected together thanks to HTTP links. These links carry no meaning about the relationship between two documents, or about the content a document exhibits. Linked Data refers to those same documents, but enriched with *semantic* meta-data. For example, the content of a document can be described by this meta-data by indicating the language it was written in, or the content's type, e.g. book or article. Tim Berners-Lee has first introduced the idea of such enriched web documents, called *Linked Data*. For such a vision to be viable as the web is now a large collection of heterogeneous data, a framework that sets the best practices for publishing and searching Linked Data is a necessity. The *Semantic Web* is such a framework and is not aimed to be a replacement of the Web as we know it, but as an extension giving relevant knowledge about documents on the web.

Over the past years, the quantity of Linked Data available has been increasing and is now reaching a climax. Semantic Data now at hand, the challenge is to be able to search for information efficiently. Linked Data is commonly stored within databases called *Triple Stores*, where querying is normally done thanks to the standardized query language SPARQL defined with the Semantic Web. However this language, while being very expressive, is known not to scale efficiently to a large amount of data. *SIREn* is an Information Retrieval-based search engine which goal is to store and to query efficiently at the web scale Linked Data. SIREn is a low level application, meant to be used by other applications such as web services that provide people information results about what they queried for. *Sindice* is a web service that provide people and machines a search engine over Linked Data through the use of SIREn.

SIREn builds an index over the Linked Data collections for searching and retrieval purpose. With the amount of Linked Data growing quickly, the size of such indexes and the performance to retrieve data from it are then points of interest to build a scalable system. A compressed index is of course important in order to take less space, however decreasing the time spent on IO operations is also important in order to increase the query throughput. Within the long studied domain of Information Retrieval, compression methods have been developed for that purpose. However they were built for traditional indexes, i.e., indexes built on collections with textual documents (unstructured data). On the contrary, SIREn builds indexes on Linked Data, i.e., (semi) structured data. Because the type of data differs, the model used to index documents is different, which implies that the distribution of indexed values are different. State of the art compression techniques are not adapted to this new distribution of values, and so a new compression class has been developed to match this need: *Adaptive Frame Of Reference*.

Index compression is one aspect that has to be taken care of when building an efficient and scalable Information Retrieval search engine. Some information in the index is not relevant to a query. Reading unnecessary data when processing a query is then wasted time. An other aspect to optimize the performance of a search engine is to avoid reading useless information with regards to a query as much as possible by using *self-indexing* techniques. The *Skip Lists* is a data structure that can be used for that purpose. Highly efficient compression techniques are nowadays block-based algorithms. However Skip Lists abstracts from the block view, discarding statistics about the block that could be used for a more efficient structure. *SkipBlock* is a block-based self-indexing model that generalizes the Skip Lists structure.

We review in this report some techniques that aim to reduce the IO access time overhead in order to increase the overall performance of the Information Retrieval web search engine. We present AFOR, a compression method that provides both fast compression and decompression speed, and yet with a high compression ratio. We propose SkipBlock, a new self-indexing model that reduces the time of random lookups from inverted lists.

1.1 Motivation

1.1.1 Semantic Web

> I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A "Semantic Web", which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize.
>
> – Tim Berners-Lee, 1999

The Semantic Web is an extension of the current Web and acts as a layer of resource descriptions on top of the current one. These descriptions are metadata, data about data, that specify various information about web resources such as their author, their creation date, the kind of the content, ... Within the Semantic Web framework, *Resource Description Format*[1] (RDF) is a standard format that is used to describe a document. RDF adds information about a web document by embedding metadata. RDF data is structured data[2]. However semantic data on the Web is large, and each dataset may have its own schema that may be partially known. They do not all follow a rigid structure as in databases. We say that the data is *semi-structured* [1].

Over the past few years, an increasing quantity of Semantic data has been published on the web, and not just by some programmers but also by industries, governments and individuals. Some recent RDF data published are:

- Governments provide their information to citizens thanks to Linked Data such as the British dataset http://data.gov.uk/, or http://data.gov/ by the USA government.

[1] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-Concepts

[2] Structured data follow a strict a format that allows machine to understand and process it.

- News agents such as the New York Times or Reuters include semantic information (i.e., RDF) in their articles.
- Web content management systems like Drupal that ease the process to expose one's data as structured information for the Semantic Web.
- BestBuy[3] which publishes product descriptions, store details and other information in a machine-readable format.

As per Tim Berners-Lee quote, the semantic web is becoming a reason to get things moving. The article[4] relates some of the possibilities the semantic web gives. It reports that it allows people to get more involved with the government, e.g. by publishing on what the public money is being spent on.

1.1.1.1 Semantic Web in Action

A lot of effort has been put in building a set of best practices to make the Semantic Web viable. This section shall give an overview of this work by first reviewing prominent ontologies that have been developed to describe knowledge for a specific domain. Finally we will show an application use case based on Sindice, called *Sig.ma*, that aggregates Linked Data from different datasets into one single "entity profile", i.e., all resources related to a same entity, for example such as the entity DERI.

1.1.1.2 Publishing Semantic Data

Tim Berners-Lee, the person credited with coining the terms Semantic Web and Linked Data, has frequently described Linked Data as "the Semantic Web done right"[5]. It aims to create different semantic data datasets of knowledge and to connect them together. The we can perform a search on a particular dataset, and then to move to an other dataset because it is somehow related to the current one. Through a 5-stars Linked Data publishing scheme, Tim Berners-Lee sets the steps to follow so that individuals and government people contribute in a good way to enrich the semantic data collection:

⋆ Available on the web (whatever format), but with an open licence

⋆⋆ Available as machine-readable structured data (e.g. excel instead of image scan of a table)

⋆⋆⋆ as (2) plus non-proprietary format (e.g. CSV instead of excel)

⋆⋆⋆⋆ All the above plus, use open standards from W3C[6], (RDF and SPARQL) to identify things, so that people can point at your stuff

⋆⋆⋆⋆⋆ All the above, plus: Link your data to other people's data to provide context

The Open Data Movement aims at making web data freely available to everyone. The goal of the Linking Open Data community project is to extend the Web with a data commons by publishing various open data sets as RDF on the Web and by setting RDF links between data items from different data sources. RDF links

[3] BestBuy: http://www.bestbuy.com/

[4] "The Web Turns 20: Linked Data Gives People Power, Part 1 of 4" article, which can be found at the address http://www.scientificamerican.com/article.cfm?id=berners-lee-linked-data

[5] slides: http://www.w3.org/2008/Talks/0617-lod-tbl/#(1)

[6] The World Wide Web Consortium, W3C: http://www.w3.org/, is an international community to build Web standards

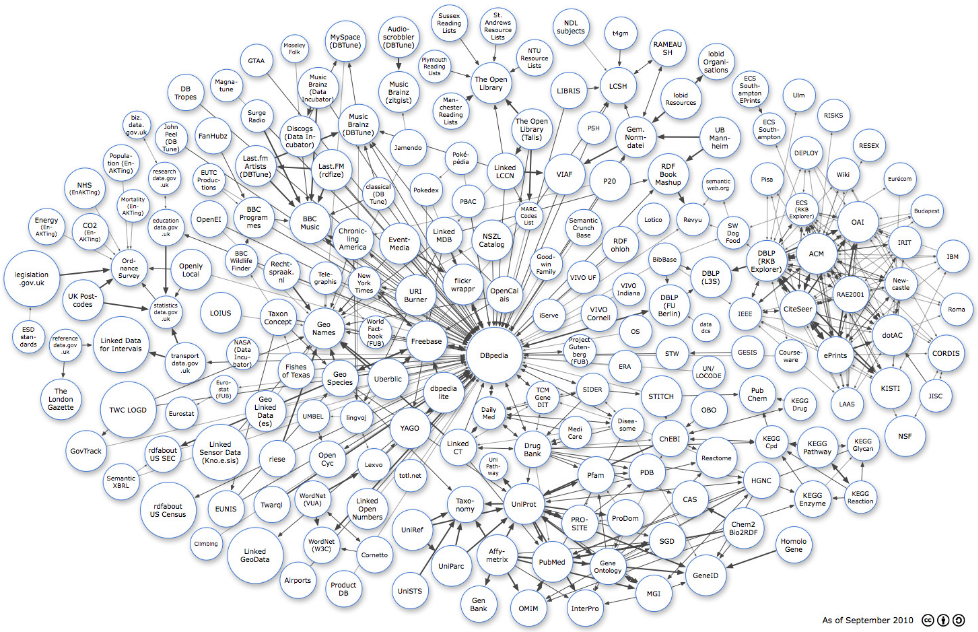


Figure 1.1: The Linking Open Data Cloud diagram, where all the current RDF datasets are presented and how they are interlinked to each other.

allow people, and even more so machines, to navigate through the heterogeneous information. Web data is represented using standard technologies, allowing data to be reused across applications. The Figure 1.1 depicts the current state of the Linking Open Data Cloud[7].

Knowledge can be modeled using a hierarchical structure of relations between the concepts of a domain, resulting into an *ontology* describing the domain. Several domains have been represented with ontologies, such as the relations between people for the social networking, the relations and descriptions of products for the e-commerce, or even chemistry knowledge in the e-biology. In this paragraph, some of the developed ontologies are reviewed.

Linked Data for social relations

FOAF the Friend-Of-A-friend project[8] is creating a Web of machine-readable pages describing people, the links between them and the things they create and do, by using the FOAF ontology. This project create files containing information about people, who use them in order to describe themselves. The FOAF file an HTML document embedding RDF thanks to RDFa[9], created using `http://foaf.me/` and describes my interests, my contact information and the people I know. By displaying this document into one's homepage, it acts as an identity profile that can be used by any other applications.

[7] `http://richard.cyganiak.de/2007/10/lod/`

[8] `FOAF:http://www.foaf-project.org/`

[9] Resource Description Framework – in – attributes (RDFa) is a W3C recommendation for embedding RDF into HTML documents. RDFa: `http://www.w3.org/TR/xhtml-rdfa-primer/`

Open Graph Protocol In the last few months an increasingly number of web sites have been using the *"I Like It"*-button provided by FaceBook which tells how many people like the thing it is attached to. This button also allows to update the profile of the person who clicked on it. The technology that is actually used is the *Open Graph Protocol*[10] which is based on the RDF model. With several RDF property that the protocol defines, a person is able to give a lot of information. For example, a picture can be further described by giving it a type (e.g. PNG or GIF), a title or even Geographic locations. This protocol defines a way thanks to the Semantic Web to represent rich data, allowing to publish rich information within social sites.

Linked Data for e-commerce GoodRelations[11] is an ontology that is used to describe precisely what a business is offering. Its purpose is to create a RDF dataset that provides information about products such as its price, its features or where it is available and so on. Recently, Google advised[12] merchants to use vocabulary to describe precisely their products, so that they can be used more efficiently in search results (e.g., related products to the one searched for, showing in Google search page results). GoodRelations is an ontology that Google advised to use for this task.

1.1.1.3 Semantic Data Mashup

Sig.ma [39] is both a service and an user application built on top of Sindice that demonstrates the power of the Semantic Web by a combined use of semantic queries, data aggregation and responsive user interaction, that together create rich entity descriptions. It can be accessed online at the address `http://sig.ma/`.

Advanced Browsing the Web of Data. Starting from a textual search, the user is presented with a rich aggregate of information about the entity likely identified with the query (e.g., a person when the input string is a person name). Queries can be about people, as well as any other entity described on the Web of Data, e.g., locations, name of documents, products, etc. As the user visualises the aggregate information about the entity, links can be followed to visualise information about related entities.

Live views on the Web of Data: rich, embeddable, addressable. At any aggregation page, Sig.ma offers rich interactions tools to expand and refine the information sources that are currently in use as well as some data oriented cleansing functionalities to hide and reorder values and properties.

As a result, it is possible to interactively create curated "views" on the Web of Data about a given entity which can be then addressed with persistent URLs, therefore passed in instant messages or emails, or embedded using a specific markup in external HTML pages. These views are "Live" and cannot be spammed: new data will appear on these views exclusively coming from the sources that the mashup creator has selected.

Structured property search for multiple entities, Sig.ma APIs. A user, but more interestingly an application, can make a request to Sig.ma for a list of properties and a list of entities. For example requesting "emails, affiliation and picture" for Stephane Campinas, and receiving updated information about them.

[10] Open Graph protocol: `http://opengraphprotocol.org/`

[11] GoodRelations: `http://www.heppnetz.de/projects/goodrelations/`

[12] Google announcement: `http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=146750`

1.1.2 Linked Data and Information Retrieval

With the amount of semantic data growing more each day, a system that is able to scale and to search efficiently over that data is crucial. Indeed in order to show the power of the semantic web with applications such as Sig.ma, it is fundamental that the data structures providing the information are highly efficient. Information Retrieval techniques have been chosen as a solution.

Information Retrieval deal with the representation and the search of textual documents. A user express his need for information by issuing a query to the machine. Information Retrieval systems are nowadays widespread with the daily use of web search engines by millions of people of Google or Bing to name a few.

An Information Retrieval search engine is known to scale well [7] to large collections such as the Web. However they are built for textual documents, i.e., unstructured data, opposed to the (semi) structured Linked Data. Building an Information Retrieval search engine for semantic data raises new challenges. Such a search engine has to deal with highly heterogeneous information sources, to offer advanced search interfaces and to provide to many simultaneous clients an access to billions of entities in millions of data sources with 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 system maintenance. *SIREn* is a search engine developed to meet such requirements.

1.2 Internship

In this section I present how this report is structured, before reporting what has been achieved during the internship.

1.2.1 Outline of this Report

The report is divided in four parts: the *Background*, the *Methods*, the *Benchmarks* and the *Additional Work and Conclusion*. In the Background part the different technologies that will be used as a base for the rest of the document are reviewed. Then in the Methods part I present the contributions that have been done during the internship. Then the Benchmarks part reports and discuss evaluation experiments on the new structures presented in the previous part. Finally some work that has been done on parallel is reported in the part “Additional Work and Conclusion”.

Part II: Background
In Chapter 3 I review the main structures and methods used in Information Retrieval. In Chapter 4 I further describe the RDF data model representation, since RDF data is the basis of the work done. in Chapter 5 I present SIREn, the search engine for semantic data using Information Retrieval techniques. In Chapter 6 the state of the art compression techniques are reviewed. The Chapter 7 presents a self-indexing structure, *Skip List*.

Part III: Methods
In Chapter 8 I introduce two compression techniques as the main results of our work. In Chapter 9 a block-based self-indexing structure is introduced, based on the Skip List.

Part IV: Benchmarks
In Chapter 10 the benchmarking framework throughout the internship in order to perform precise evaluations is described. In Chapter 11 benefits of our new compression technique are reported,

with further experiments on the scalability of the system based on that compression technique in Chapter 12. In Chapter 13 the efficiency of the block-based self-indexing structure is discussed.

Additional Work and Conclusion
In Chapter 14 I present the work that has been done in parallel on SIREn. In Chapter 15 I recall what are the main achievements for this internship and what the future will hold for the author.

1.2.2 Contribution of the Internship

During this internship, the work first consisted in develop ping a new high performance compression technique, *AFOR*, which is more adapted for compressing data produced by a system like SIREn. Then we worked on developing a new self-indexing structure, *SkipBlock*, which allows faster document lookups from the index. A short paper that Renaud Delbru and myself wrote entitled "SkipBlock: Self-Indexing for Block-Based Inverted List", which can viewed at the Appendix B, has been accepted for an oral presentation at the *The* 33^{rd} *European Conference on Information Retrieval* (ECIR)[13] conference. This conference is an European forum for the presentation of new research in the field of Information Retrieval.

[13] ECIR: http://www.ecir2011.dcu.ie/

Chapter 2

Digital Enterprise Research Institute

DERI is a worldwide organisation of research institutes with the common objective of integrating semantics into computer science and understanding how semantics can improve computer engineering in order to develop information systems collaborating on a global scale. A major step in this project is the realisation of the Semantic Web.

2.1 DERI - International

DERI International is constituted of four research institutes. DERI Innsbruck, located at the Leopold-Franzens University in Austria, and DERI Galway, located at the National University of Ireland Galway in Ireland, are the two founding members and key players. DERI Stanford and DERI Korea are representative members of DERI in their country and are research institutes that have joined DERI International. DERI performs academic research and leads many projects in the Semantic Web and Semantic Web Service field. DERI has been successfully acquiring large European research projects in the Semantic Web area such as DIP[1] (Data, Integration and Processes) or Nepomuk[2] (Semantic Web desktop). DERI collaborates with several large industrial partners as HP, ILOG, IBM and CISCO but also with medium-sized and small industrial enterprises. DERI is aware of industry requirements and maintains close relationships with industrial partners in order to validate research results and transfer them to industry. DERI also has many research partners, such as the W3C, FZI Karlsruhe or École Polytechnique Fédérale de Lausanne (EPFL).

2.2 DERI - Galway

DERI Galway was founded in June 2003 by prof.dr. Dieter Fensel and is currently managed by prof.dr. Stefan Decker. DERI Galway is attached to the National University of Ireland Galway (NUIG). DERI Galway currently has 130 members composed of senior researchers, PhD students, master and bachelor students, management staffs and interns. DERI is a Centre for Science and Engineering Technology (CSET) funded principally by the Science Foundation Ireland (SFI) but also by Enterprise Ireland, the Information Society Technologies (EU) and the Irish Research Council for the Humanities and Social Sciences. DERI divides its research into three main domains

[1] DIP: http://dip.semanticweb.org/

[2] Nepomuk: http://nepomuk.semanticdesktop.org

1. Social Semantic Information Spaces
2. Semantic Reality
3. Application Oriented Research Domain

Within these research strands individual units focus on one core competency for realising DERI's mission and its work with its industrial partners. Each unit specialises in a particular research discipline that has relevance for realising the overall goals of the institute.

2.3 The Mission

The Web is an area in constant progress, and a major step is on its way: linking together the real work with the virtual world. A first revolution was done with the apparition of social networking. This marked an important change in the way people have been using the Internet: a massive amount of data are now available and shared. Individuals as well as enterprises gain from it. However the use of this data is limited because they are platform specific (e.g. LinkedIn, Facebook, Myspace, ...). There are *islands of information* that do not link to each other.

DERI's mission is to research new technologies that will realise the link between the real and the virtual worlds (Figure 2.1): the development of applications using the semantics of data, software that interconnects the islands of information. Two main steps are necessary to achieve this link:

1. *sensors* that collect data from the physical world: temperature sensors for home automation, body sensors such as the heartbeats to better help people control their health.
2. *semantic spaces* that break down barriers between information allowing their thorough use.

These axes lead to the realisation of linking the real world with the virtual world, creating what DERI calls the *semantic reality* [38].

2.3.1 DI2 - Data Intensive Infrastructures

DI2 position itself within DERI as a unit which purpose is to develop systems and infrastructures in order to make Social Semantic Spaces a possibility. The requirements to analyse and understand the inner workings and underlying fundamental concepts of the Web along with the elevated scalability requirements for the development of new Web infrastructures have demonstrated the necessity of Data Intensive Supercomputing (DISC) infrastructures to support researchers and developers. A prominent characteristic of DISC (as promoted by major players such as Google, Yahoo and MSN), as opposed to classic supercomputing, is the importance of very advanced data management software over high-end hardware configurations. DISC data entries are in fact known to be formed by hundreds or thousands of commodity machines connected using common commercial networking infrastructures. It is then up to the software to be able to deliver high capacity, throughput, scalability, re-configurability and fault tolerance. To be able to conduct credible research and development in the Web domain, DERI requires a DISC infrastructure. As DISC in itself is the subject of ongoing research, the goal of this work programme is to advance the research and applications of data intensive infrastructures, to create, maintain and offer to researchers a state-of-the-art DISC infrastructure, and to research novel algorithms and data structures for scalable handling of large amounts of semantic information.

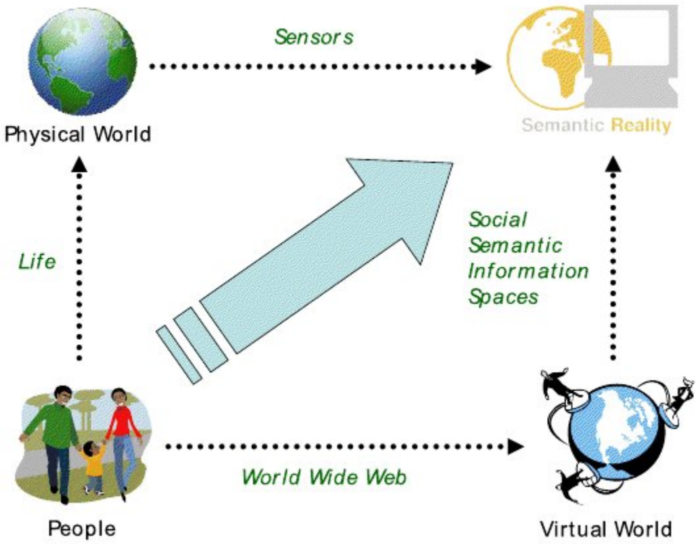


Figure 2.1: Semantic reality.

2.4 Working Environment

My internship lasted from February 2010 to December 2010 within the Data Intensive Infrastructures (DI2) unit. *Sig.ma* is a semantic web application based on Sindice, the web service that provides search and retrieval capabilities over semantic data. At the core Sindice uses SIREn, a search engine based on Information Retrieval, for the purpose of retrieving semantic data. My supervisors were Renaud Delbru, Ph.D student at the time but graduated to doctor on December, and Dr. Giovanni Tummarello. The internship consisted to assist Renaud Delbru on his thesis about SIREn.

My work was closely watched by my supervisors, with a regular meeting between R.Delbru and myself in order to discuss the progress, new research ideas, design and implementations. The research was performed based on other research scientific publications, in order to provide a solid background for our own research.

On the hardware aspect, DERI lent me a laptop for the duration of the internship. Also I had access to DERI servers so that I could perform my own experiments and benchmarks on a sane environment.

As for the programming languages used during the internship, JAVA and scripting languages such as shell scripts, ruby, python and sed were used.

Part II

Background

Chapter 3

Information Retrieval

Information Retrieval [14] (IR) deals with representing, searching and manipulating large collections of texts. The Figure 3.1 depicts the components of an IR system. By issuing a *query* which consists in a group of *keywords* (i.e., index terms) to an IR system, a user expresses his need for information to the machine, i.e. the *information need*. A *term* is not necessarily a word, but can also be a phrase, a date or any other set of words. The machine task is to return a set of documents that are useful, i.e., *relevant*, to the user in some sense. A returned document is given a *score* with regards to the issued query, that indicates how much relevant the document is to the query. This way returned results can be *ranked* with regards to their scores.

The use of IR services is now widespread thanks to web search engines such as Google of Bing. Users of such services expect it to return up-to-date, accurate and near-instantaneous answers to a query. [7] reports that IR web search engines are still able to give good performance even at the web scale, which contains billions of documents. In order to achieve a sub-second response to the user, web search engines do not only rely on hardware performance, but also on the efficiency of data structures and other models that it will use.

A major task of a search engine is to maintain and manipulate an *inverted index* for a document collection. This data structure is the main structure used by a search engine for searching and ranking. As a basic function, the inverted index provides a mapping between the terms and the locations in the documents in which they occur. Terms' Locations are stored into a data structure called *inverted lists*. The size of inverted lists are on the same magnitude as the collection itself. Thus search and retrieval operations must be done with care in order to be efficient.

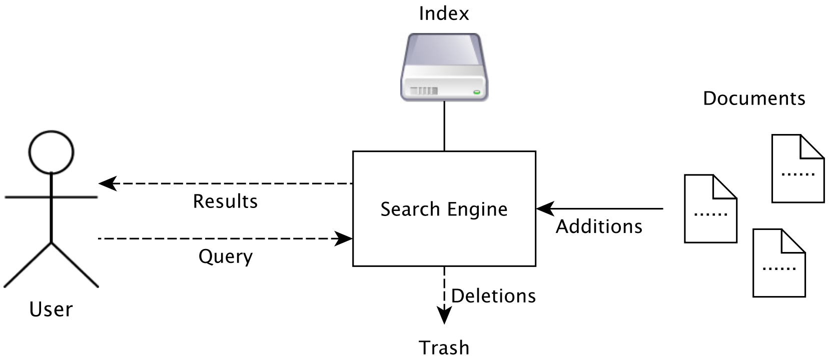


Figure 3.1: Principal components of an IR system.

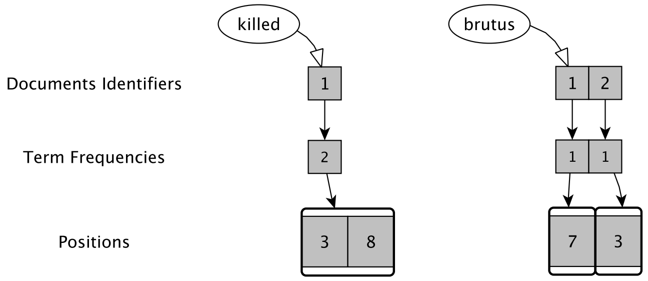


Figure 3.2: Inverted lists for the terms "killed" and "brutus" in documents from the Table 3.1. Each inverted list stores the documents identifiers, the term frequencies and the positions information.

3.1 Inverted Index

An index can be seen as a matrix with the terms (e.g. unigram) for rows and the collection's documents as columns. In this matrix, an element equal to one if the term i occurs in document j, and zero if not. However such a matrix contains practically only zeros, thus a lot of wasted space. Inverted List is a basic IR structure that reduces such a matrix to only store ones' values. All the inverted lists taken together form the inverted index.

From the collection a dictionary of the words appearing in is created. The dictionary's terms have been first filtered, by removing the punctuation for example. Given a term from the dictionary, a linked list is associated to it and stores all documents records where the term occurs in. A document record can be simply a serial number of the document (i.e., document identifier), but additional information like the term frequency and the position can be given. The term frequency corresponds to the number of occurrences of the term in a document, and the position indicate the positions of the occurrences within that document. The Figure 3.2 depicts two inverted lists storing documents identifiers as integers, term frequencies and positions. These inverted lists are built using the terms "killed" and "brutus" from documents in the Table 3.1. There is a relation one-to-one between documents identifiers and term frequencies, and a relation many-to-many with the positions. There are as much position values as there are occurrences (i.e., term frequency) of the term in the document. The documents identifiers are stored in increasing order in order to process queries, as presented in Section 3.2.

3.1.1 Indexing

In this section, traditional inverted index construction is presented, then the block-based construction which makes possible to handle large collection. Finally I discuss the importance of compression for large collection, and present a commonly used in Information Retrieval encoding technique.

An inverted index is the group of inverted lists built from a collection. The basic steps to create an inverted index in the main memory from a collection of documents are:

1. make a first pass through the collection to gather term-document-i dentifiers pairs
2. sort the pairs with the term value as the key
3. group together docIDs of a same term into an inverted list.

These steps are reported in Table 3.1 with two documents taken from the "Shakespeare's Collected Works", storing only the document identifier (docID) in the inverted lists and the document frequency (DF), i.e.

Document 1: I was killed i' the Capitol; Brutus killed me. **Document 2:** The noble Brutus hath told you Caesar was ambitious.

term	**docID**		**term (sorted)**	**docID**		**term**	**DF**		**Inverted List**
I	1		ambitious	2		ambitious	1	↦	1
was	1		brutus	1		brutus	2	↦	1 → 2
killed	1		brutus	2		capitol	1	↦	1
i'	1		capitol	1		caesar	1	↦	1
the	1		caesar	2		hath	1	↦	1
capitol	1		hath	2		I	1	↦	1
brutus	1		I	1		i'	1	↦	1
killed	1		i'	1		killed	1	↦	1
me	1	⟹	killed	1	⟹	me	1	↦	1
the	2		killed	1		noble	1	↦	1
noble	2		me	1		the	2	↦	1 → 2
brutus	2		noble	2		told	1	↦	1
hath	2		the	1		you	1	↦	1
told	2		the	2		was	2	↦	1 → 2
you	2		told	2					
caesar	2		you	2					
was	2		was	1					
ambitious	2		was	2					

Table 3.1: Inverted index creation from two documents taken in "Shakespeare's Collected Works". Each word is filtered and associated to its document identifier (docID). After sorting the pairs on the term, the inverted lists are created for the set of terms. The inverted lists store only the docID of the document the term appears in.

the number of documents the term appears in. Each words in the documents are filtered by removing the punctuation and lowering the case. The set of terms defines the dictionary of the inverted index. For each of the dictionary, an inverted list is then created.

In order to build an inverted index once and for all, the in-memory inverted index is written to the disk into what is called an *inverted file*. The Figure 3.3 summarize the steps from the collection to the creation of an inverted file. The inverted file contains all the inverted lists, written one after the other. The dictionary is also stored in that same file, where each term possess information about the inverted list it points to: the inverted list's offset or the document frequency, i.e., the number of documents the term appears in.

3.1.1.1 Merge-Based Indexing

The previous indexing technique works in-memory, and then cannot handle large collection with large inverted lists. Merge-based indexing is a technique that divides an inverted index construction into segments, which are then merged together to form a complete inverted index. Thus an inverted list can span over multiple segments. When the main memory is filled or that a threshold has been reached, a segment is written to a secondary storage space (e.g. the disk) into an inverted file. Each on-disk inverted file contains an inverted index independent from the one in an other segment. When merging inverted files, it is then necessary to update inverted lists of a same term but stored in different segments. The Figure 3.4 depicts the merge process of three inverted files into one. An inverted list spans over these files: the documents identifiers values are then local to each file. Thus it is necessary to update these identifiers numbers in order to keep an ordered inverted list after merging.

Such an index construction requires two steps to finish. The first one consists in building multiple

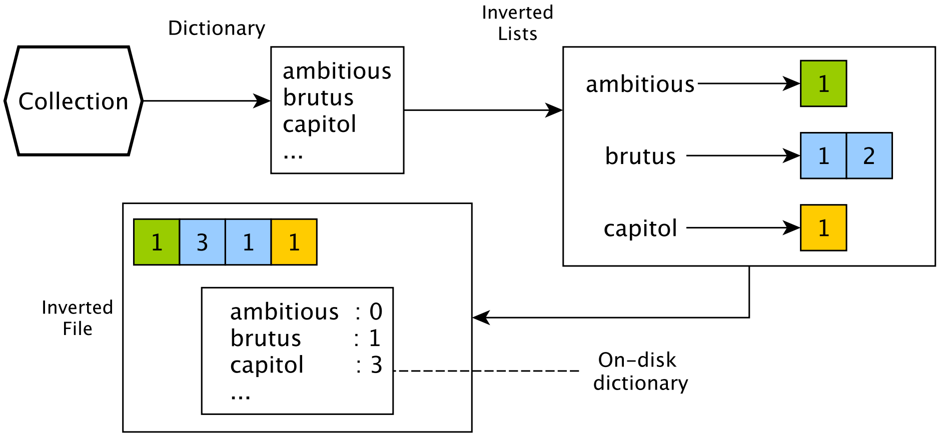


Figure 3.3: Indexing, from a collection of documents to the inverted file. The in-memory inverted lists are written to disk one after the other. The terms of the embedded dictionary possess offset to the associated inverted list in the file.

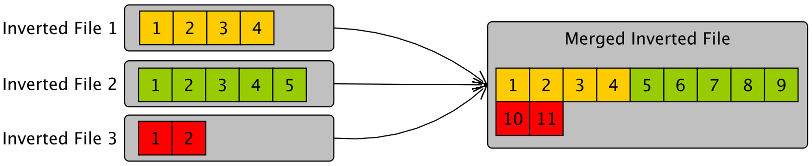


Figure 3.4: Merge-based indexing of three inverted files. An inverted list, storing documents identifiers, spans over these files. When merging these files into one inverted file, the documents identifiers are updated to keep an ordered list.

segments and is called the *commit* step. The second one consists in the merging process of multiple files into one inverted file and is called the *optimization* step. These steps are believed to be a common operation for incremental inverted index.

3.1.1.2 Block-Based Inverted List

For performance and compression efficiency, it is best to store separately each data stream of an inverted list [5]. In a non-interleaved index organisation, the inverted index is composed of three inverted files, one for each stream of values (i.e., documents identifiers, frequencies and positions). Each inverted file stores contiguously one type of list, and three pointers are associated to each term in the lexicon, one pointer to the beginning of the list in each inverted file.

An inverted file is partitioned into blocks, each block containing a fixed number of integers as shown in Figure 3.5. 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 meta-data 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.

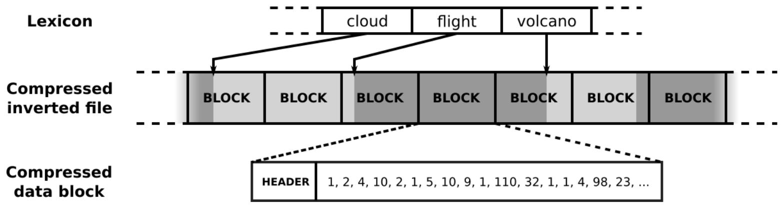


Figure 3.5: Inverted index structure. Each lexicon entry (term) contains a pointer to the beginning of its inverted list in the compressed inverted file. An inverted file is divided into blocks of equal size, each block containing the same number of values.

3.1.1.3 Index Updates Strategies

Updating the inverted index with new documents is done via two approaches: by *batches* or *incrementally*. The update strategy by batches stores added documents into a buffer, and writes them to the index on-disk when some threshold is reached. While this is adapted to static indexes, in other words indexes that do not need to be updated right away. However in some cases such as Internet news search engines, the user would expect the index to be updated as soon new documents are detected. This use case is matched by incremental updates strategy. With this strategy, the index is composed of the ones on disk, but also with an index built in-memory. This second index holds the updated documents temporarily that are bound to be written to the index on-disk when a threshold is reached. The question of deleted documents is handled thanks to a table that stores the documents flagged in deleted state.

3.1.2 Delta-Encoding

Large collections result in large inverted files. As these files are stored on disk, compressing their data is then crucial in order to save storage space. *Delta-encoding* is a basic encoding commonly used in Information Retrieval which aims to decrease the size of inverted lists, thus reducing inverted files disk space.

With an ordered and increasing list of integers, it is possible to considerably reduce the size by storing not the actual integers values but the gaps. A gap is the difference between two consecutive integers. As the list is increasing, the difference between an integer at position i and an integer at position $i - 1$ will always be positive. To decompress a value at position i from a delta-compressed list, we just have to sum up every values up to i. The Figure 3.6 depicts a delta-compressed list of integers. By storing gaps values, we can reduce the size of the list from 88 to 38 bits.

Documents identifiers are stored in increasing order in each inverted list, and can then be delta-encoded. Also the positions values are stored in increasing order, but they are local to a document. Thus position values relative to a same document are delta-encoded, and the encoding is re-initialized when changing documents.

3.2 Query Model

A query is an user request for information about the collection. Documents matching that query are returned back to the user. In this section, I present the query model used in Information Retrieval and explain how keyword and Phrase queries are processed.

The model used to represent documents is the *vector space model*, commonly called the “bag of words” model. In this model documents and queries are seen as vectors, where a component represent a term in the

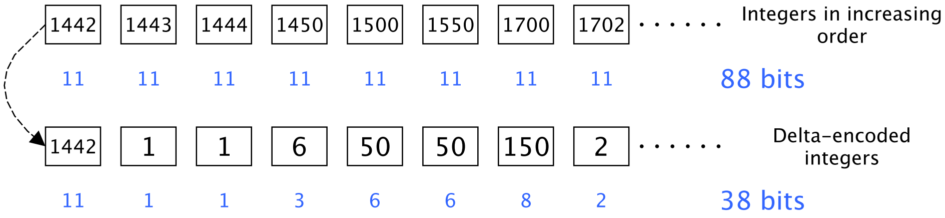


Figure 3.6: Delta-encoding of an increasing ordered list of integers, showing the size in bits of the two lists.

collection. Given a query and a set of document vectors, we compute the angle between them. The smaller this angle is, the higher the document will be ranked with regards to the query.

The query model is a Boolean model, where terms that are to appear in the returned documents are combined together using Boolean operators. Boolean operators include two binary operators, i.e., *AND* and *OR*, and one unary operator *NOT*. Their operands can be either a term or a Boolean sequence of terms. With this model, documents in the indexed collection are nothing more than a set of terms. Set theory operations are used to interpret such queries. The Boolean operators are binary operations and their operands are set of words. The And operator is interpreted as an *Conjunction*, the OR operator as an *Disjunction* and the NOT operator as the *Exclusion*. A simple Boolean query on the two documents from the Table 3.1

brutus *AND* ambitious

searches for all documents containing the terms brutus and ambitious.

In this model, a document is seen as a *bag of words*, where the exact ordering of terms is ignored but the number of occurrences of each term (i.e. term frequency) is material.

3.2.1 Keyword Query Processing

Keyword queries search for documents containing the terms in the Boolean expression. Their processing follows the algorithm

1. Retrieve inverted lists of all the query's terms.

2. Apply set operations, i.e., union, intersection or set difference, on the inverted lists.

3. Returns documents mapping to the documents identifiers in the inverted list, resulting of the set operations.

The former query "brutus AND ambitious" returns the document identifier 1, as the intersection of their inverted list give only this one identifier. The Figure 3.7 depicts the processing of that query, showing in red the documents identifiers occurring in both inverted lists.

3.2.2 Phrase Query Processing

The position of terms in a document is very important, as it can change the meaning of these terms in the document. For example the keyword query "*stanford AND university*" may return a document containing the sentence "The inventor Stanford Ovshinsky never went to university". However the given query suggested to search information about a University in Stanford. The returned document is then irrelevant to the query,

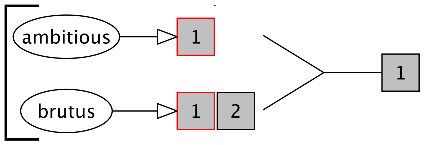


Figure 3.7: Processing of the keyword query "brutus AND ambitious". The only document containing both terms is the document 1, from the Table 3.1.

to what really is needed by the user. In order to perform more precise search, the position information about the terms in documents is necessary. Indeed restricting the search to documents having the keyword terms appear consecutively one after the other will discard irrelevant documents. As four terms are between Stanford and University in the former example, such a document will not be returned if restricting to zero words in-between. The query that takes into account the position of terms within documents are called *Phrase queries*.

A *n-gram* is a sub-sequence of n terms in the document. For instance, a bi-gram (i.e., a sub-sequence of two terms) in document 1 in Table 3.1 can be "I was" or "Brutus killed". There exist two ways to handle phrase queries. The first one relies on indexing n-gram occurring in the collection. Indexing bi-gram would follow the same process as in the Table 3.1, but with bi-gram instead of single terms. However it restricts to bi-gram only, and to search for tri-gram (i.e., three terms in the sub-sequence) it would mean to index also tri-gram. This is highly inefficient as a huge storage space is then needed. The second way uses the position information of terms within documents to process phrase queries. Thus only uni-gram (i.e., one term in the sub-sequence) are indexed with the relative position of each term. Also we can handle this way phrase queries with any number of terms in the sub-sequence.

3.2.2.1 Phrase queries processing algorithm

The first step in processing phrase queries is the same as processing keyword queries: after retrieving inverted lists of each term, we perform operations on the inverted lists to get the list of documents identifiers matching the terms in the query. The second step consists in filtering these documents according to the position information. The algorithm to discard or not a document works on the positions of keywords occurring in a same document, and is as follows:

Let R be the list of documents identifiers that results from the first step.

1. For each document i in R.

2. Retrieve the list $P_{T1,i}$ of positions for the term T1 in document i, and the list $P_{T2,i}$ of positions for the term T2 in document i.

3. Compare position values $p_1 \in P_{T1,i}$ and $p_2 \in P_{T2,i}$

   (a) The absolute difference between p_1 and p_2 is *greater than* 1 ⇒ within the list with the lower value, we advance until reading a position greater than the other value.

   (b) The absolute difference between p_1 and p_2 is *equal to* 1 ⇒ the current document is kept and we go to the next document in R.

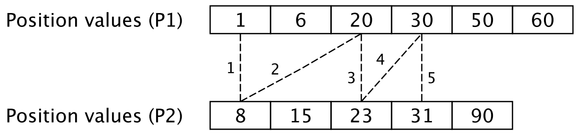


Figure 3.8: Documents filtering based on the position of terms. P1 and P2 are two positions lists of two different terms occurring in the same document. Dashed lines represent comparison between position values. The order in which these comparisons are made is indicate by their label number.

The Figure 3.8 depicts with dashed lines the comparisons performed in the step 3 in the former algorithm. We first compare 1 and 8 and keep on reading values in P1 until the value 20. Then the comparison between 20 and 8, and so on. The algorithm stops at comparison 5, as the difference is equal to 1. The document is thus kept in the results of the phrase query. In the Table 3.1, the phrase query “brutus killed” would match the document 1, as the terms appear respectively in positions 7 and 8. We can note that as the difference computed in steps 3 is absolute, the phrase query “killed brutus” also matches that document. This reflects the “bag of words” query model in Information Retrieval.

Chapter 4

Semantic Web

The World Wide Web has drastically changed over the past decade. Tim Berners-Lee is the person who introduced the idea of linked information thanks to HyperText Markup Language (HTTP), which is the core of the web. The expectations Tim Berners-Lee had of a global information space, where any data can be read, written and shared is now concrete. The challenge is now to organize this massive amount of knowledge and to improve the interactions people have with this huge and complex system.

A major problem of the web comes directly from its core, the HTML. Thanks to HTML, documents can be linked together, and a human can move from one document to an other with those links. A computer is able to interpret this language, however the content of documents is written in natural language, and is then only accessible to people.

Most of the content on the web is only readable by people. One can read the information of documents and understand its meaning. On the opposite, computers only know to which documents a document is related to thanks to HTML markups. However they do not carry descriptions about the content of a document and how two documents are related to each other.

This is the reason why Internet applications like search engines have limited capabilities. A search engine looks for a term T in a document, but it cannot search for related documents, as it lacks a formal description of the content. A search engine help people look for data, but in the end to find precise information, he has to himself follow HTML links. This is a highly consuming task, with limited results.

A solution is to provide computers with comprehensible Web data is to describe documents and their relationships with meaningful metadata. The Semantic Web aims to merge web resources with machine-understandable data to enable people and computers to work in cooperation and to simplify the sharing and handling of large quantities of information.

In this chapter I present the vision of the Semantic Web, before describing the various technologies in the Semantic Web. Then I present how Web data is represented to make it machine-understandable. Finally I present some projects that use Semantic Web technologies.

4.1 The Framework

The Semantic Web defines a set of technologies which are a standard on how to describe, manipulate and query Semantic data to have in the end a solid base of machine-readable knowledge. The Semantic Web presents a layer hierarchy (Figure 4.1), where each layer has its own purpose. A layer of this Semantic Web Stack uses the information provided by layers below. This shows how Semantic Web is made possible and

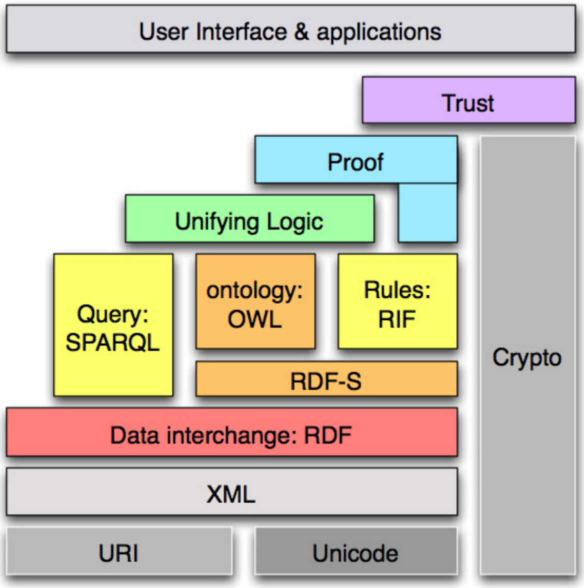


Figure 4.1: The Semantic Web Stack. It presents each layer of the Semantic Web, where each have a defined a purpose and that make the Semantic extension of the Web as we know it possible.

how it is not a replacement of the current Web but an extension.

Uniform Resource Identifier (URI) provides means for uniquely identifying semantic web resources.

Unicode[1] serves to represent and manipulate text in many languages. Semantic Web should also help to bridge documents in different human languages, so it should be able to represent them.

XML extensible Markup Language (XML) is the standard syntax that enables creation of documents composed of structured data. Semantic web gives meaning (semantics) to structured data.

Resource Description Framework (RDF) is the standardized representation of meta-data.

RDF Schema (RDFS) provides basic vocabulary for RDF.

Web Ontology Language (OWL) extends RDFS and allows a more complex representation of resources (i.e., ontology). It gives then the possibility of more advanced reasoning over data.

SPARQL is a RDF query language. It can be used to query any RDF-based data and provide information to Semantic Web applications.

Logic-Proof a reasoning layer that infers new knowledge from the data.

Trust prove the trustfulness of the data thanks to cryptographic techniques.

User Interface is the final layer that will enable humans to use semantic web applications.

URI is a string that is used to identify resources over the Internet. It is often used by markup languages to specify external documents or a specific section of that document. For instance *http://di2.deri.ie/* is an URI that describes the Data Intensive Infrastructure unit.

An ontology in Information Science is a formal representation of knowledge as a set of concepts within a domain, and the relationships between those concepts. It is used to reason about the entities within that domain, and may be used to describe the domain. For example we can have a simple ontology describing animals like rats and cats are mammals; a mammal is an animal. Such an ontology can be used when searching for terms in different domains.

4.2 RDF: Resource Description Framework

RDF is a framework that allows data on the web to be shared and reused across applications. It provides features that facilitate data to be interchange between datasets, even if they have different ontology. RDF is used to describes things in the Internet, and the relations between things. It extends the linking structure of the Web to use URIs to name the relationship between concepts or objects as well as the two ends of the link. This linking structure of the Web forms a directed, labeled graph called *RDF graph*, where the edges represent the relationship between two objects, the graph nodes.

4.2.1 RDF Model

RDF is a standard model[2] for data interchange on the Web. Two connected nodes and the named relation form a *triple*. The set of triples form the RDF graph. The three parts that compose a triple are:

1. a *subject*: the resource that this triple describes.

2. an *object*: the resource that the subject is related to.

3. a *predicate*: the description of the link between the subject and the object.

The Figure 4.2a depicts a triple as a node-arc-node link. The direction of the arc is significant: it always points toward the object. The nodes of an RDF graph are its subjects and objects. The meaning given to a RDF graph is the conjunction of all its triples.

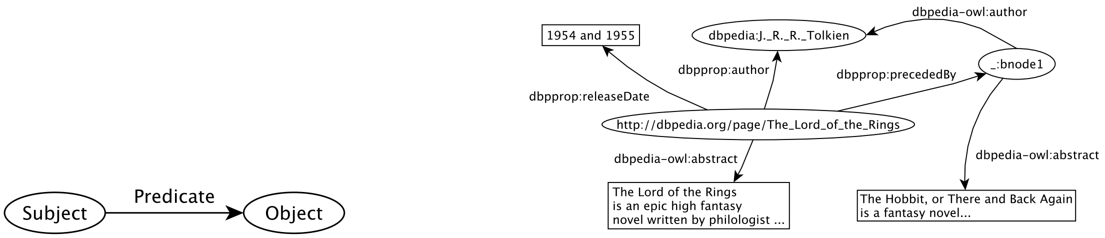
A RDF graph defines three types of nodes:

Literal a literal is a simple string, used to describe dates, titles or any text in natural language. A literal may be *plain* or *typed*:

- A plain literal is a string combined with an optional language tag.
- A typed literal is a string combined with a datatype URI.

URI reference (URIref) a URIref is a Unicode string used in a RDF graph to represent URIs, and it can have an optional fragment identifier which is indicated by the character “#”. For example the URI reference http://www.w3.org/2000/01/rdf-schema#comment shows the URI before #, and the fragment “comment” that indicates which identifier to use within the URI. This particular fragment “comment” is used to describe the subject resource.

[2] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-Concepts



(a) A triple

(b) A RDF graph describing the work “The Lord of the Ring” by J.R.R Tolkien, using several QNames.

Figure 4.2: The RDF model used to represent knowledge.

Blank Node a blank node refers to resources that are not identified.

In a triple, the subject can be either a URIref or a blank node. The predicate must be a URIref, while the object can be either one of the three node types.

The URI tends to be very long, which becomes inconvenient when using frequently a same URIref. QName is a name space used as an abbreviation of a URIref. Its syntax is *URI:fragment*. For example the fragment “comment” within the URI `http://www.w3.org/2000/01/rdf-schema#` can be shortened to *rdfs:comment*, where rdfs is mapped to the URI.

In the drawing convention of a RDF graph, URIrefs and blank nodes are drawn with an ellipse, and literals with a rectangle. The Figure 4.2b depicts a RDF graph describing the work “The Lord of the Rings” written by J.R.R Tolkien. In this graph, the QNames “dbpprop” and “dbpedia-owl” map respectively to *`http://dbpedia.org/property/`* and *`http://dbpedia.org/ontology/`*. This graph provides three different triples describing the sequel. The RDF graph gives the meaning that the resource `http://dbpedia.org/page/The_Lord_of_the_Rings` has for author J.R.R Tolkien, that it was released in the years 1954-55 and that a portion of the English abstract is “The Lord of the Rings is an epic…”. We also know that an unknown resource preceding “The Lord of the Rings” novel was also written by Tolkien.

4.2.2 RDF Serialization

Many serialization format for RDF have been developed, such as RDFa[3] which is used to embed RDF triples within XML documents. However during my internship only the serialization format *N-Triple*[4] has been actively used, and so it will be the only one described here. Using the N-Triple format, the triples within a RDF graph can be serialized with a line-based plain text format. It is a format recommended to interchange data between different applications.

This serialization uses an Extended Backus-Naur Form grammar, where each triple is written into a line, ending with a point. The format is ***Subject Predicate Object .*** . A URIref is written enclosed between the characters ’<’ ’>’. The literals are written between double quotes, followed by an optional language tag *@language* for plain literals (e.g. “bird”@en), and by *’^^’URIref* for typed literal (e.g. “abc”^^<`http://example.org/datatype1`>, where abc is of type datatype1). A blank node is represented by *’_’:name*, where name is an internal identifier which allows to have several blank nodes within a RDF graph. The RDF

[3] `http://www.w3.org/TR/xhtml-rdfa-primer/`

[4] `http://www.w3.org/TR/rdf-testcases/#ntriples`

<TLR>	<dbpprop:author>	<dbpedia:J._R._R._Tolkien>	.
<TLR>	<dbpprop:releaseDate>	“1954 and 1955”	.
<TLR>	<dbpedia-owl:abstract>	“The Lord of the Rings is an epic…”	.
<TLR>	<dbpprop:precededBy>	$\_ : bnode1$	.
$\_ : bnode1$	<dbpedia-owl:author>	<dbpedia:J._R._R._Tolkien>	.
$\_ : bnode1$	<dbpedia-owl:abstract>	”The Hobbit, or There and Back Again is a fantasy novel…”	.

Table 4.1: N-Triple representation of the RDF graph in Figure 4.2b.

graph in the Figure 4.2b can be serialized in N-Triples format as in Table 4.1 where *TLR* refers to the URI http://dbpedia.org/page/The_Lord_of_the_Rings.

4.2.3 SPARQL

The RDF model introduce a mean to represent data and any type of data can, at least to some degree, be represented in this model. SPARQL[5] is a standardized query language for RDF. Designed so that queries can be expressed across diverse data, SPARQL is able to express not only conjunction and disjunction of RDF graphs but also complex queries, e.g. by using supplementary binary operators such as arithmetic operators and comparisons in various filters.

A SPARQL query contains a set of *triple patterns* that matches a sub-graph of RDF graphs. Triple patterns are expressed in the same way as RDF triples, i.e., subject, predicate, object, the difference being that any part of that triple can be a variable. 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 sub-graphs matching precisely the graph pattern defined in the query. For instance, the SPARQL query

```
PREFIX dbpprop: <http://dbpedia.org/property/>
SELECT ?abstract
WHERE
{

    <TLR> <dbpprop:author>  ?abstract  .

}
```

has a single triple pattern that matches the first row of the triples in the Table 4.1. The first line shows the QName used in the query.

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

Chapter 5

SIREn

People are using more and more Semantic Web technologies and the quantity of Semantic data will keep on expanding. Scalable and performant RDF management systems are then becoming a necessity for semantic applications in order to use thoroughly semantic data. Moreover traditional search engines do not use the structured information available with semantic data. Taking the e-commerce as an example, looking for a particular product is done by entering keyword queries into the search engine and web pages are returned. However these documents are most of the time irrelevant as they only include the key words, but are not necessarily about the product itself. Also such web pages are likely to provide information about other unrelated products as well. What is really looked for then is the *entity* of the product and anything that can be related to that entity.

Sindice[1] is a web service which purpose is to offer efficient searching capabilities over the semantic data. Sindice is built on top of the Semantic Information Retrieval Engine, SIREn[2], a system based on Information Retrieval which goal is to search not for web pages but for "entities".

In this chapter, the indexing model of SIREn is presented before introducing its query model.

5.1 Related Work

In this section, we describe the current approaches for indexing RDF data, before reviewing the existing search models for (semi) structured data.

5.1.1 Retrieval of RDF Data

The growth of RDF data incurs the need of efficient and scalable data management systems. Several systems have been proposed based on Relational DataBases Management Systems (RDBMS) [8]. RDF data management systems are able to store a large amount of data and use SPARQL as a query language which allows to perform complex searches over the collection. A user is able to perform complex request when the database schema is known. However given the heterogeneous data collection on the web, the added value of SPARQL is limited. Indeed it is difficult to write precise queries since the data structure and the ontology may vary from one dataset to an other.

[1] Sindice: http://sindice.com/

[2] SIREn: http://siren.sindice.com/

Other approaches [24, 19, 26, 15] 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 easier 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 Semplore [41]. Semplore has extended inverted index to encode RDF graph approximations and to support keyword-based tree-shaped queries over RDF graphs. However the increase of query capabilities comes at a cost, and the scalability of the system becomes limited.

5.1.2 Search Models for (semi) Structured Data

In addition to standardised query languages such as SPARQL, a large number of search models for semi-structured data have been developed. Some of them focus on searching structured databases [3, 10, 27, 32], XML documents [17, 25, 33] or graph-based data [2, 28, 30] 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 [29, 34, 41, 20, 22] have extended simple keyword-based search with structured queries capabilities. [20, 22] propose a partial solution to the lack of expressiveness of the keyword-based approach by allowing search using conditions on attributes and values. [29, 34, 41] 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 [29, 34, 41] are not applicable on a very large scale.

The search model introduced in Section 5.2 is similar to [20, 22], 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.

5.2 Formal Model

The expansion of Semantic Web applications has increased considerably and it will continue on this progression. A wide range of different technologies has been developed to represent and use semantic data. For the purpose of covering a majority of data sources when searching for entities, a formal model has been introduced. In this section, I give a summary of a labeled directed graph model, before describing its query model.

5.2.1 Entity Attribute-Value Model

Semantic data can be used to describe any kind of data, from a physical person to an abstract notion. These *entities* are at the core of any data and are the unit that is to be searched for with SIREn. This section presents fundamentals about the Entity Attribute-Value model, a generic model which goal is to support scenarios involving entities.

A Semantic graph (e.g. a RDF graph) is a graph that gives information about several entities somehow related to each other. This graph can then be split into sub-graphs, each one describing a particular entity. A *star graph*, i.e., an entity at the center with associated incoming and outgoing relations. For instance, the

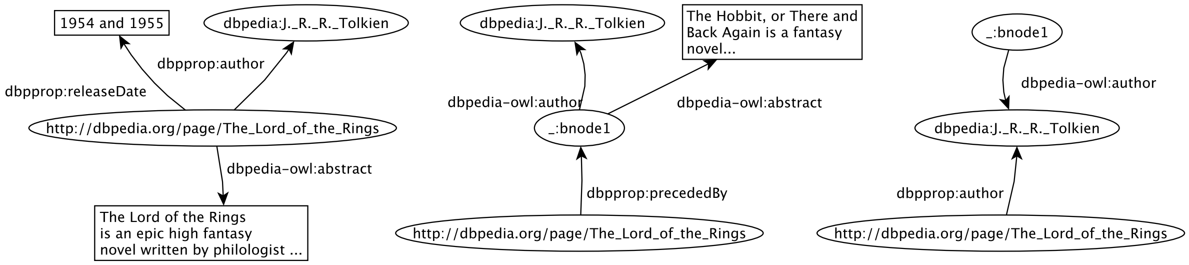


Figure 5.1: A representation of the RDF graph in Figure 4.2b divided into three entities: *The Lord of the Rings*, *_:bnode1* and *Tolkien*.

RDF graph in Figure 4.2b can be split into three entities, as depicted in the Figure 5.1. The graph describing "The Lord of the Rings" work is divided into the `http://dbpedia.org/page/The_Lord_of_the_Rings`, *_:bnode1* and *dbpedia:J._R._R._Tolkien* entities. The star graph gives information about the entity and is then generically called an *entity description*.

A *dataset* is a source of semantic data, such as a domain or a site web and it gives the *context* of the data it contains. An *attribute* refers to the relation between an entity and an object, e.g. the predicate in a RDF triple. For example "dbpedia-owl:author", "dbpedia-owl:abstract" and "dbpprop:precededBy" are the attributes associated with the _:bnode1 entity. An attribute can also be *multi-valued* as a book can have multiple authors for instance. A *value* refers to the node related to an entity.

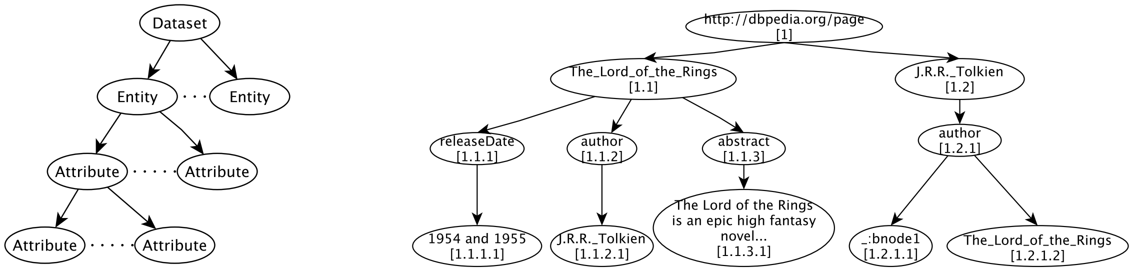
5.2.1.1 Node-Labelled Tree Model for RDF

SIREn uses a node-labelled tree structure to capture the relationship between values, attributes, entities and datasets. Taking the RDF data model (the triple) as a support, the tree represent four kinds of nodes as depicted in the Figure 5.2a:

- the *dataset* that indicates the context of the data.
- the *entity* (the subject).
- the *attribute* (the predicate).
- the *value* (the object).

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.

A node-labelled tree model enables to encode and efficiently establish relationships between the nodes of a tree. The two main types of relations are Parent-Child and Ancestor-Descendant. To support these relations, the requirement is to assign *unique identifiers*, called node labels, that encode the relationships between the nodes. The Dewey encoding [9] is a simple node labelling scheme. 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). The Figure 5.2b depicts the tree representation of the RDF graph in Figure 4.2b using the Dewey encoding. The dataset is the domain of the URI the graph is hosted at. In that representation,



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

(b) Node-labelled tree model with Dewey encoding of the dataset in Figure 4.2b.

Figure 5.2: The node-labelled tree model.

the qualified names (i.e., QNames) have been removed in order to keep only the property name, for clarity purpose since the property has the same semantic even if the QNames are different (e.g., dbpprop:author and dbpedia-owl:author). For example the entity "The_Lord_of_the_Rings" with vector [1.1], we can find that it is a parent of the value node "J.R.R._Tolkien" with vector is [1.1.2.1].

5.2.2 Query Model

Entities present a semi-structured structural view, and not a "bag of words" as in traditional search engines. Therefore the query model search entities with Boolean combinations of attribute-value pairs.

SIREn supports three different types of queries:

full-text: traditional keyword query, useful when the schema of the dataset is unknown;

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

semi-structural: 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.

These query types provide different level of complexity, depending on the knowledge of the data structure by the user. But the SIREn engine is aimed to be used by other machines, and the possibility of complex queries are then needed. The Figure 5.3 depicts a star-shaped query that matches the "The Lord of the Rings" entity from the Figure 5.1. This semi-structural query makes use of full-text search through the use of keywords (e.g. "tolkien" or "lord"). We can also embedded Boolean search inside nodes, then the matching of entities having the keywords "lord" and "rings" are marked as a positive match for the value of an abstract (or label) attribute. The wildcard can also be used to describe unknown relations between nodes. This search model is developed to find entities that best match an entity description pattern (e.g. star-shaped pattern).

In opposite to the full-text logical view where words are seen are seen as a single bag of words, in the semi-structured view, the words are assigned to multiple distinct bag of words, one for each value and attribute. Consequently, it is possible to distinguish when words occur in a same value or different values and avoid false-positive answers.

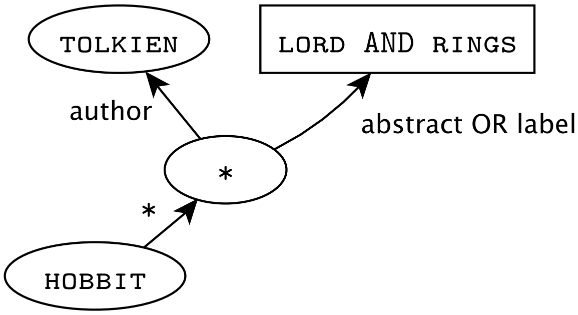


Figure 5.3: A star-shaped query matching the entity "The Lord of the Rings" from the Figure 5.1. The character * stands for the wildcard variable.

Entity	N-Triples			
TLR				
	<TLR>	<dbpprop:author>	<dbpedia:J._R._R._Tolkien>	.
	<TLR>	<dbpprop:releaseDate>	"1954 and 1955"	.
	<TLR>	<dbpedia-owl:abstract>	"The Lord of the Rings is an epic..."	.
_:bnode1				
	<TLR>	<dbpprop:precededBy>	_:bnode1	.
	_:bnode1	<dbpedia-owl:author>	<dbpedia:J._R._R._Tolkien>	.
	_:bnode1	<dbpedia-owl:abstract>	"The Hobbit, or There and Back Again is a fantasy novel..."	.
dbpedia:J._R._R._Tolkien				
	_:bnode1	<dbpedia-owl:author>	<dbpedia:J._R._R._Tolkien>	.
	<TLR>	<dbpprop:author>	<dbpedia:J._R._R._Tolkien>	.

Table 5.1: Entity-centric indexing on "The Lord of the Rings" RDF graph of Figure 4.2b. Each entity is reported with its star graph in the N-Triples format.

5.3 Model Implementation

This section aims to present the implementation of the Entity Attribute-Value model and of the query model over an inverted index. An overview of the inverted index construction is presented with the entity-centric indexing before describing more thoroughly the indexing structure.

5.3.1 Entity-Centric Indexing

The search unit of the SIREn engine is an entity. As such, SIREn aims to index entities (i.e., star graphs), and not documents as the Section 3.1.1 presents it. With a *document-centric* indexing, every terms belong to a same document. For instance the RDF graph in Figure 4.2b can be taken as a whole document, since all data has been taken from the same web page `http://dbpedia.org/page/The_Lord_of_the_Rings`. However when searching for the entity *The Lord of the Rings*, we just want information about this entity and not also about any related work. This is a reason why SIREn uses an *entity-centric* indexing.

With a document-centric indexing, all the terms from the N-Triples of the Figure 4.2b are seen as one document. Then any queries on one of those terms will return that document. With an entity-centric approach, we index the three different entities separately as reported in the Table 5.1.

SIREn stores triples documents into a table where each documents are stored as rows, a column having a meaning for the document (e.g. title, abstract, ...). The Table 5.2a reports a document-centric strategy for

< s >	< p1 >	< o1 >	.
< s >	< p1 >	< o2 >	.
< s >	< p1 >	< o3 >	.
< s >	< p2 >	< o4 >	.

(a) Document-centric indexing.

< p1 >	< o1 >	< o2 >	< o3 >	.
< p2 >	< o4 >	.		

(b) Entity-centric indexing.

Table 5.2: Two different indexing strategies for semantic data, e.g. RDF triples. The characters "s", "p" and "o" refer respectively to the subject, predicate and object in a triple.

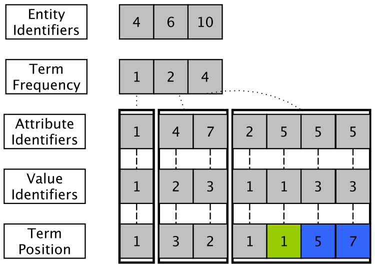


Figure 5.4: Inverted Lists with the SIREn model.

semantic data like RDF triples, where the first column stores the triple's subject, the second one the predicate and the third one the object. As an attribute can be multi-valued, a lot of space is wasted as three rows are used to store three triples where only the object changes. In order to save space, thus improving indexing and querying performances since less data to be written and read, SIREn uses a representation format called *N-Tuples*. It consists to collapse all triples with a same predicate into one row and to remove the subject cell, as depicted in the Table 5.2b.

5.3.2 Inverted Index Structure

Traditionally in Information Retrieval-based search engines, there are three streams of integers that form an inverted list. A stream of documents identifiers, a stream of term frequencies and a stream of positions. In SIREn there are five stream of integers, being streams of entity identifiers, of term frequencies, of attributes identifiers, of values identifiers and of positions, identifiers taken from the Dewey encoding depicted in the Figure 5.2b. The term frequency corresponds to the number of the term's occurrences in an entity description (e.g. a star graph). The term position corresponds to the relative position of the term within its node.

Because of the Dewey encoding, the identifiers numerical values have a clustering effect: the entity identifier is local to a dataset, and the attribute identifier is local to that entity, and so on. The Figure 5.4 depicts the five stream of identifiers that together form the inverted list in SIREn. As an example for the locality of identifiers values, the position 5 and 7 in the entity 10 (i.e., blue squares) describe a same term in a same value node (i.e., value node with identifier 3). However the green square is local to the same attribute 5, but not to the same value node.

The repetitive structure of semantic data (e.g. a same URI in the predicate cell) implies that combining the entity-centric indexing with a delta encoding of the inverted lists result in a compact inverted index structure. Indeed because of the locality of the values, the gap between integers will be small.

Chapter 6

Inverted List Compression

The inverted lists data, which are needed to process queries, are stored on disk. This is the reason why reading the data is very costly on the queries performance. The more data has to be read, and the slower the query response will be. Thus inverted lists are compressed in order to reduce the number of bytes read at query time, apart from the obvious goal to reduce disk space consuming.

In this chapter, I present a state of the art of compression algorithms. While Variable Byte is an algorithm that works on one integer at a time, Rice, Simple family algorithms and Frame Of Reference-based techniques are block-based. Given a block of integers from the list to compress, they aim to compress it as best as possible using different approaches.

6.1 Variable Byte

Variable Byte (VByte) is an easy to implement algorithm, which makes it a commonly used compression algorithm. VByte is a byte-aligned, i.e compression at the byte level, algorithm that compress lists of values one integer at a time. A compressed byte is divided in two parts

1. the most significant bit is a *flag* bit.

2. the 7 bits left are used to store the integer.

For an integer v to be compressed, the lower 7 bits are stored in one byte. The flag is put at (a) 1 if there is still bits remaining in v or (b) 0 otherwise. This process is repeated until no more bits are left. The Table 6.1 reports the algorithms used for VByte. The Algorithm 1 compress an array L of integers into an array LC of bytes. The loop from line 3 to 7 continues as long as there are still 7 bits left int the current integer int, where it outputs the 7 lower bits of int into LC and put the flag at 1. The Algorithm 2 decompress all the array LC. The line 5 to 9 loops as long as the flag is at 1, meaning that the next byte data belongs to a same integer value.

This algorithm presents the positive aspects of an easy implementation and of good compression and decompression overall. However this algorithm possesses two drawbacks. The branching condition leads to branch mispredictions which makes it slower than CPU optimised techniques such as *Frame Of Reference*-based algorithms presented next. Moreover, VByte has a poor compression ratio since it requires one full byte to encode small integers (i.e., $\forall n < 2^7$).

Algorithm 1: Compression algorithm.

```
Input: An array L of integers
Output: An array LC of bytes
$i \leftarrow 0$
for $int \in L$ do
    while $(int \,\&\, 127) \neq 0$ do
        $LC[i] \leftarrow (int \,\&\, 127) \mid 128$
        $x \leftarrow int \gg 7$
        $i \leftarrow i + 1$
    end
    $LC[i] \leftarrow int$
end
```

Algorithm 2: Decompression algorithm. The functions `len` returns the size of an array, and `poll` retrieves and removes the first element of an array.

```
Input: An array LC of bytes
Output: An array L of integers
for $i$ to len($L$) do
    $b \leftarrow$ poll($LC$)
    $dec \leftarrow b \,\&\, 127$
    $shift \leftarrow 7$
    while $(b \,\&\, 128 = 1)$ do
        $b \leftarrow$ poll($LC$)
        $dec \leftarrow dec \mid ((b \,\&\, 127) \ll shift)$
        $shift \leftarrow shift + 7$
    end
end
```

Table 6.1: Variable Byte algorithms. L denotes the array to be compressed, LC its compressed array. The Algorithm 2 decompress the values in LC, returned by the Algorithm 1.

6.2 Rice

In Rice, 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 store in binary format using b bits. In unary format, a integer n is represented with n consecutive bits at 1, and a final bit at 0, serving as the termination criterion. 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 advantages of Rice is to achieve a very good compression ratio. However, it is in general the slowest method in term 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 very demanding in CPU cycles.

6.3 Simple 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 (S-64) based on 64-bit machine words [6]. In the experiments, we report only S-64 results since its overall performance was always superior to Simple9 [4]. 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. The Table 6.2 reports the 14 configurations used in S-64. For instance, 12 integers of 5 bits each at maximum can be encoded into a machine word using the Status 4. The *Wasted Bits* row highlights the problem encountered with some configurations, as some bits are left unused with a straight forward approach. A solution is to allow the last integer of the configuration to use these extra bits.

On top of providing a good compression ratio, decompression is done efficiently by reading one machine

Status	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Bits	1	2	3	4	5	6	7	8	10	12	15	20	30	60
Group	60	30	20	15	12	10	8	7	6	5	4	3	2	1
WastedBits	0	0	0	0	0	0	4	4	0	0	0	0	0	0

Table 6.2: Status options used in Simple-64 coding scheme. Bits refers to the number of bits an integer is coded with. Group refers to the number of integers that can be put into one machine word. WastedBits reports the number of bits wasted for each status.

word and by using a precomputed lookup table over the status bits in order to execute the appropriate optimised routine (one routine per configuration) to decode the data bits using shift and mask operations only. However, Simple coding performs one table lookup per machine word which costs more CPU cycles than the other highly CPU optimised techniques presented next.

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 at each new window.

6.4 Frame Of Reference

Frame Of Reference (FOR) determines the range of possible values in a sub-block, called a *frame*, and maps each value into this range by storing just enough bits to distinguish the values [23]. 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 [45, 6] is to choose as frame the range $[0, max]$ where max is the largest number in the group of delta values.[1]

Given a frame $[0, max]$, FOR needs $\lceil \log_2(max + 1) \rceil$ bits, called a *bit frame*, to encode each integer in a block. 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 [45] which avoid branching conditions. Each routine is loop-unrolled to encode or decode m values using shift and mask operations only. Listing 6.1 and 6.2 show the routines to encode or decode 8 integers with a bit frame of 3. There is a compression and decompression routines 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

[1] 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.

reads the bit frame, performs one table lookup to select the decompression routine and executes iteratively the routine over the compressed sub-blocks.

```
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 6.1: 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 6.2: Loop unrolled decompression routine that decodes 8 integers represented by 3 bits each

6.5 Patched Frame Of Reference

Patched Frame Of Reference (PFOR) [45] is an extension of FOR less vulnerable to outliers in the value distribution. PFOR stores outliers as exceptions such that the frame of reference $[0, max]$ is greatly reduce. PFOR first determines the smallest max value 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 block 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* [45] is created.

For large blocks, the linked list approach for keeping track of the position of the exceptions is costly when exceptions are sparse since a large number of compulsory exceptions has to be created. [45] proposes to use blocks of 128 integers to minimise the effect. [43] proposes a non-compulsive approach where the exceptions are stored along with their offset in the second block section. We choose the latest approach since it has been shown to provide better performance [43].

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 complex heuristics to find the best bit frame and set of exceptions for a block.

Chapter 7

Self-Indexing Techniques

When intersecting two or more inverted lists, we often need to access random records in those lists. The basic approach is to scan linearly the lists to find them. Such an operation is not optimal and can be reduced to sub-linear complexity in average by the use of the self-indexing approach [36].

This Chapter presents a self-indexing technique, *Skip Lists*, which is a probabilistic alternative to balanced trees. While being easier to implement and to optimize than the former structure, the Skip Lists structure still provides a logarithm access time to records.

7.1 Related Work

The Skip Lists data structure is introduced by [37] as a probabilistic alternative to balanced trees and is shown in [18] to be as elegant as and easier to use than binary search trees. Such a structure is later employed for self-indexing of inverted lists in [36]. Self-indexing inverted list enables a sub-linear complexity in average when intersecting two inverted lists. [11] proposes a way to compress efficiently a Skip Lists directly into an inverted list and shows that it is possible to achieve a substantial performance improvement. however by embedding the skips directly into the inverted list disrupts the values distribution of the list. Indeed block-based compression methods show performance dependent on the integers distribution. [16], the authors introduce a method to place skips optimally given the knowledge of the query distribution. However the query distribution can not be known in the use case of Sindice, because of the diversity of datasets schema it is impossible to create a set of queries that likely will be run. [35] presents a generalized Skip Lists data structure for concurrent operations.

7.2 Skip Lists

Skip Lists is a self-indexing structure that builds a sparse index over the inverted lists and provides fast record lookups. In this section, we first present the Skip Lists model and its associated search algorithm. We finally discuss the effect of the probabilistic parameter with respect to the Skip Lists data structure and search complexity.

7.2.1 The Skip Lists Model

Skip Lists are used to index records in an inverted list at regular interval. These indexing points, called *synchronization points*, are organized into a hierarchy of linked lists, where a linked list at level $i + 1$ has a probability p to index a record of the linked list at level i. The probabilistic parameter p is fixed in advance and indicates the *interval* between each synchronization point at each level. For example in Figure 7.1, a synchronization point is created every $\frac{1}{p^1} = 16$ records at level 1, every $\frac{1}{p^2} = 256$ records at level 2, and so on. In addition to the pointer to the next synchronization point on a same level, a synchronization point at level $i + 1$ has a pointer to the same synchronization point at level i. For example in Figure 7.1, the first synchronization point at level 3 (i.e., for the record 4096) has a pointer to the level 2 which has itself a pointer to the level 1. This column of synchronization points is called a *tower*. This hierarchical structure enables to quickly find a given record using a top-down search strategy.

Given the probabilistic parameter p and the size n of an inverted list, we can deduce two characteristics of the resulting Skip Lists data structure:

1. the expected number of levels

$$L(n) = \left\lfloor \ln_{\frac{1}{p}}(n) \right\rfloor \tag{7.1}$$

2. the size, i.e., the total number of synchronization points is given by

$$S(n) = \sum_{i=1}^{L(n)} \lfloor n \times p^i \rfloor \tag{7.2}$$

which sums up the number of synchronization points expected at each level.

$L(n)$ sets the maximum number of levels, since [37] shows that the probability that it is actually greater is very low.

The number of levels at a synchronization point is fixed according to its position in the inverted list. If this was set randomly, a level i might not have the probability of expected synchronization points at p^i. This reflects that the balancing of synchronization points is probabilistic rather than strictly enforced, thus making the algorithm to build the structure easier than in balanced trees. For example at the building phase of the Skip Lists in the Figure 7.1, writing the 256-th record triggers the building of $L(256) = 2$ levels.

7.2.2 Search Algorithm

The Skip Lists enables to retrieve an interval containing the target record and is performed using a top-down strategy. The search within an interval is discussed in Section 9.2. The search walk starts at the head of the top list, and performs a linear walk on a level as long as the target is greater than a synchronization point. The walk goes down one level if and only if the target is lower than the current synchronisation point. The search finishes when the current synchronization point is (a) equal to the target, or (b) on the bottom level and greater than the target. The reached interval then contains the target.

The search complexity is defined by the number of steps necessary to find the interval holding the target. In the worst case, the number of steps at each level is at most $\frac{1}{p}$ in at most $L(n)$ levels. Consequently, the search complexity is

$$C_S = \frac{L(n)}{p} \tag{7.3}$$

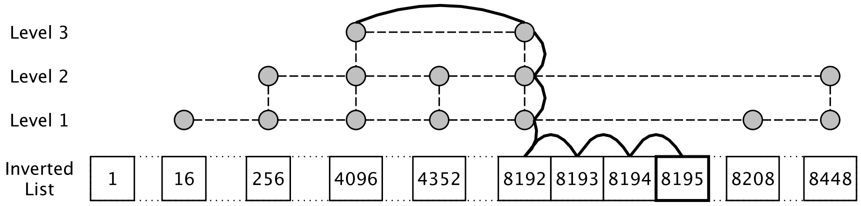


Figure 7.1: Skip Lists with $p = \frac{1}{16}$. Dashed lines denote pointers between synchronization points. The solid line shows the search path to the record 8195.

The Figure 7.1 depicts with a solid line the search path in a Skip Lists with $p = \frac{1}{16}$ and $L(n) = 3$ levels to the record 8195. At the top of the Skip Lists, we walk to the record 8192. Then we go down to level 1 and stop because the current synchronization point (i.e., the record 8208) is greater than the target. At this point, we know that the target record is in the interval directly before the current synchronization point on the inverted list. The searched record is reached by scanning the records until the target is found.

7.3 Implementation

In this section we present the implementation of the Skip Lists structure as found in the open source project Apache Lucene[1].

Structure The structure's data is kept on disk, and as such its implementation is done so that the cost of IO operations are reduced. Each level of the Skip Lists is stored as one stream, in order to read continuous data on disk when reading synchronization points at a same level. The levels are stored one after the other in reverse order, starting from the top level to level 1. The reverse order storage of the levels is so that it maps the search algorithm flow, i.e. we start on the top level, then descend levels to the bottom. The Figure 7.2 depicts the Skip Lists' Lucene implementation of a synchronization point with 3 levels. The ID refers to the record identifier needed to advance in the Skip Lists. The *Data* holds information about the inverted list built upon such as file pointers. For a same synchronization point such as in the figure, the Data block stores the same information at each level about the inverted lists (i.e., duplicate data). The next grey block depicts a file pointer, intern to the Skip Lists. This file pointer stores the offset of a same synchronization point at the level below. We can point out that for performance reasons, this file pointer points to the end of a Data block, so that a duplicate information is not read twice when descending levels.

Search algorithm The algorithm flow of the Skip Lists is highly optimized by having the less branching conditions possible in order to maximizes the search throughput. Given a targeted record that possibly is in the inverted list, the *#skipTo()* method returns the interval that would hold it. On the second line, the levels streams are loaded from disk if not done already. The lines 4 to 6 walks up the levels as long as the current record identifier of level i (i.e., stored within the *currentIDs* variable) is lower than the target. This allows to read only the levels that might skip to the target. The loop starting at line 8 advances within the Skip Lists, by reading a synchronization point at level i (line 10) as long as the current identifier is lower

[1] Apache Lucene: http://lucene.apache.org/

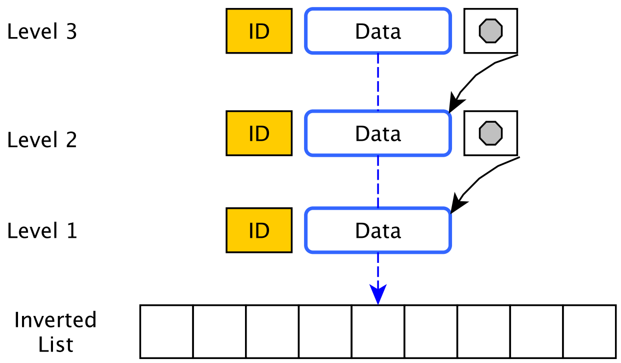


Figure 7.2: Skip Lists structure's implementation.

than the target. On lines 12-13 we advance on the stream below to the synchronization point which is the last being lower than the target.

```
 1  skipTo(int target)
 2    loadLevels();
 3    // Walk up the levels
 4    int level = 0;
 5    while (level < MAX_LEVELS - 1 && target > currentIDs[level + 1])
 6      level++;
 7    // Search for the interval containing the targeted record
 8    while (level >= 0) {
 9      if (target > currentIDs[level])
10        readSynchronizationPoint(level)
11      else
12        moveToSynchronizationPoint(level-1);
13        level--;
14    }
```

Listing 7.1: Implementation of the Skip Lists search algorithm

Part III

Methods

Chapter 8

Inverted Lists Compression For SIREn

The SIREn indexing model generates a specific distribution of values. Since each stream of values are local to an other (i.e attributes identifiers are local to an entity, values identifiers are local to an attribute, and positions are local to a value), a new compression algorithm that is better adapted to the SIREn use case was researched.

The performance aspects looked after are (1) the compression (2) the decompression and (3) the compression ratio. The decompression speed is important when processing queries because it means that less time is to be spent on IO operations. The compression ratio is important as per the storage space, but not only. As previously, less data read from disk means less time wasted on IO operations. The compression speed aspect does not matter when processing queries, but takes an important place when updating the index. Indeed SIREn possesses an incremental index update policy, thus compression algorithm showing good decompression speed will allow indexes to be updated faster than an algorithm with slower compression algorithm.

In this chapter, two of the compression techniques researched on are presented, with *RiceFOR* first, an algorithm that fuses together the Rice and FOR algorithms, and the *Adaptive Frame Of Reference* (AFOR), a new algorithm based on FOR, which goal is to decrease the FOR sensibility to outliers.

8.1 RiceFOR

Rice is an algorithm that shows compression ratio performance [44] close to the interpolative coding, which is an algorithm that provides the best compression ratio. However its complexity is so high that it is used only as a comparison basis between algorithms. RiceFOR is an attempt to take the best from both Rice and FOR algorithms: the compression ratio of Rice and the highly CPU optimised FOR execution flow.

8.1.1 Algorithm

In the one hand, Rice computes the average of a block of values. Thanks to a parameter b being the power of 2 closest to this average, it computes the Euclidean division of each values by b. The remainder is compressed using b bits, and the quotient is encoded into an unary format. On the other hand, FOR compress a block of values by applying a compression routine on sub-blocks of 32 integers, where the values across the sub-blocks are encoded into a same number of bits. By computing the remainders in Rice all together, we achieve the same use case as FOR, if the number of remainders is a multiple of 32. The FOR algorithm is only used to compress with CPU optimized routines the remainders. The compression ratio is the one that we can have thanks to Rice.

The RiceFOR algorithm works as follows. First we compute the remainders given a parameter b of all values to be compressed. We might have to add some fake values (i.e., zeros) if the block's size is not a multiple of 32. These remainders are outputted into a temporary block that will be compressed using the FOR algorithm with b for the bit frame. Then the quotient of all values are to be encoded with the unary format, as they would have been with the original Rice algorithm.

The decompression algorithm works in the same way as the original Rice, the difference being that the remainders are to be decompressed with FOR.

8.1.2 Implementation

Rice provides a good compression ratio [44], however the computation of the statistics and unary encoding slow its compression and decompression speed. Along with the optimization provided by the AFOR highly performing routines, two optimized algorithms have been developed in order to encode and decode efficiently values in unary format: an optimized routine to encode a value into unary format, and a routine to decode unary values at the byte level inspired from the Rice implementation in [42].

Compression Using FOR to compress the remainders computed for the Rice algorithm, RiceFOR is implemented as follows:

Given a block B of integers

1. the power of 2 directly lower to the mean of B is used as the bit frame.
2. each sub-blocks of 32 remainders are compressed with that bit frame using the FOR routines such as the one presented in the Listings 6.1.
3. the quotient of each integer (with the division by the bit frame) are encoded in unary format, using the optimized routine reported in the Listings 8.1.

Decompression The remainders are decompressed with the FOR routines, and the quotients are decoded using a byte-based decoding method presented in the Listings 8.2. Indeed a byte is likely to encode multiple values in unary format (e.g. 4 ones are encoded in one byte as 0101 0101), thus it is possible to decode multiple values by reading a byte a single time.

In the method *encodeUnary* presented in the Listings 8.1, the loop encode the quotient q into the array of byte b of compressed data. The variable *bit* is global and records the number of bit already compressed. On line 3, the local variable *r* holds the number of bits in the current byte that are still unused. The variable *ones* holds the number of bits that will be set to 1. The switch structure displays 8 hard coded cases to write from 1 to 8 bits at 1. With the example of case 3 on line 7, the current byte, i.e., $bit/8$, has 3 bits (7 in binary format: 111) appended to it. These operations will repeat until the quotient q has been completely encoded. The termination criterion of the unary encoding, i.e., bit at zero, is computed on line 15. This suggests that the current byte has been previously initialized to zero.

The method presented in the Listings 8.2 reads a byte containing values in unary format and decode them into an int array, starting at the position *offset*. All the unary encoded values are quotient computed by the Rice algorithm, with a same parameter *frameBit*. The switch structure contains all 256 possibilities of bits configurations, and hard codes for each cases the corresponding updates. For example the 26 case, which in binary base equals 0001 1010, encodes 5 values which respectively are from the least to the most significant bit 0, 1, 2, 0 and 0. Thus the quotients are respectively 0, $1^{bitFrame}$, 2×1^{bitFrame}, 0 and 0, with $bitFrame$ the divisor parameter used in Rice.

```
encodeUnary(int q, byte[] b)
  while (q > 0) {
    int r = 8 - (bit % 8);
    int ones = r > v ? v : r;
    switch (ones) {
      ...
      case 3:
        b[bit / 8] |= (7 << (bit % 8));
        bit += 3;
        break;
      ...
    }
    q -= ones;
  }
  bit += 1
```

Listing 8.1: Unary format encoding.

```
decodeUnary(byte b, int[] i)
  switch (b) {
    ...
    case 26: // 26 = 0001 1010
      i[offset] += 0;
      i[offset + 1] += 1^frameBit;
      i[offset + 2] += 2*1^frameBit;
      i[offset + 3] += 0;
      i[offset + 4] += 0;
      offset += 5;
      break;
    ...
  }
```

Listing 8.2: Byte-based unary format decoding.

8.2 Adaptive Frame Of Reference

The Adaptive Frame Of Reference (AFOR) attempts to retain the best of FOR, i.e., a very efficient compression and decompression using highly-optimised routines to avoid branching conditions, 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.

8.2.1 Algorithm

AFOR extends the FOR algorithm and runs as follows. Given a block B of n integers, AFOR partitions it into m distinct frames and encodes each frame using highly-optimised routines similarly to FOR. Each frame is independent from one an other, i.e., each one has its own *bit frame*, and each one encodes a variable numbers of values. This is depicted in Figure 8.1 by *AFOR-2*. AFOR encodes along with the frame the associated bit frame with respect to a given encoder, e.g., a binary encoder. In fact, AFOR encodes (respectively decodes) a block of values by:

1. encoding (respectively decoding) the bit frame;
2. selecting the compression (respectively decompression) routine associated to the bit frame;
3. encoding (respectively 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 [40]. If a frame is too large, the encoding becomes more sensitive to outliers and wastes bits by using a too big 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 numbers of bit frames. The appropriate strategy is to rely

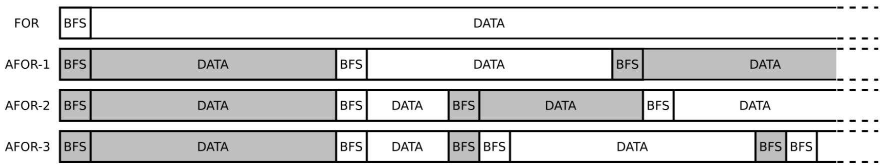


Figure 8.1: Block compression comparison between FOR and AFOR. We alternate colours to differentiate independent frames. AFOR-1 denotes a first implementation of AFOR using a fixed frame length. AFOR-2 denotes a second implementation of AFOR using variable frame lengths. AFOR-3 denotes a third implementation using variable frame lengths and the frame stripping technique. *BFS* denotes the byte storing the bit frame selector associated to the next frame.

1. on large frames in the presence of a homogeneous sequence of values.

2. on small frames in the presence of outliers.

Also, alternating between large and small frames is not only important for achieving high compression ratio but also for achieving high performance. If the frames are too small, the encoding has to perform more branching conditions to select the appropriate routine for each bit frame, and therefore the compression and decompression performance decrease. Therefore, it is better to rely on large frames instead of multiple smaller frames when it is possible. Our solution uses a greedy local optimization algorithm which is explained in the next section.

8.2.2 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 integers, there are 1.18×10^{30} possible combinations. While such a combinatorial problem can be solved via Dynamic Programming algorithms [40], 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 largest frame, therefore it greatly impacts the compression performance. Since we are not only interested by a fast decompression speed and a high compression ratio, but also by a fast compression speed, this approach is not used in our experiments. Instead, we use a local optimization algorithm that provides a satisfactory compression rate while being efficient to efficient to compute.

Local Optimization algorithm 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 first 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: $[32], [16, 16], [16, 8, 8], [8, 16, 8], [8, 8, 16], [8, 8, 8, 8]$. Then, given a list of possible configurations for a window, we estimate the size of each configuration by doing one pass over the values of the window as shown in the Algorithm 3 (Lines 1-5). This first step performs $w \times k$ comparisons with k the number of possible configurations. The second step, Lines 6-12 in Algorithm 3, performs a pass over the possible configuration and estimates the size of each configuration in order to find the best one. The `EstimateSize` function computes the cost of encoding the window given one 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: $4 + 8 \times \sum_{i=1...4} log(b_i + 1)$, where $8 \times \sum_{i=1...4} log(b_i + 1)$ is the size of the four encoded frames and 4 is the overhead to store the four bit frames.

This simple algorithm is efficient to compute, in particular if the window size is small and the frame lengths are restricted to a few number. However, it is easy to see that such method does not provide the optimal configuration for a complete block. There is a trade-off between optimal partitioning and complexity of the algorithm. However, one can possibly use a more complex method for achieving higher compression ratio if the compression speed is not critical. We decided to use this method since in our experiment we found that a small window size of 32 values and three frame lengths, 32, 16 and 8, were providing satisfactory results in term of compression speed and compression ratio. In the next Section is presented a more detailed explanation of the AFOR implementation.

Algorithm 3: 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 $i \leftarrow 0$ to $\frac{w}{l}$ do
    for $j \leftarrow i \times l$ to $(i + 1) \times l$ do
        bitFrames[i] $\leftarrow \max($bitFrames[i], $\lceil \log_2(W[j] + 1) \rceil)$;
    end
end
bestSize $\leftarrow MaxSize$;
foreach configuration c of the possible configurations do
    if EstimateSize(c, bitFrames) $<$ bestSize then
        bestSize $\leftarrow$ EstimateSize(c);
        bestConf $\leftarrow$ c;
    end
end
```

8.2.3 Frame Stripping

In an inverted list, it is common to encounter a long sequence of 1 to encode, i.e., the delta gap value. 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 widely present 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 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.

8.2.4 Frame Skipping

Block-based compression techniques encode the configuration a block of integers has been compressed with. When the length of the compressed block is unknown, this compression header possesses information about the block that can be used to skip this block and so not to decompress it.

With the AFOR algorithm, this method enables the possibility to skip a frame by reading a BFS. This is not possible with techniques such as Rice, VByte or PFOR. Since the BFS encodes the bit frame used to compress a frame, the size in bytes of a compressed frame is found by the formula

$$\mid F \mid \times BFS$$

where $\mid F \mid$ is the length of a frame. For instance the bit frame 1 (i.e., 1 bit) encodes a frame of 32 integers into 4 bytes.

With self-indexing structures (presented in Chapter **??**), the difference in bytes between two points in a stream is necessary in order to know at which position data has to be read. These *file pointers* have to be stored, taking a considerable space. Thanks to the frame skipping method, it is possible to get rid of the additional file pointers information, since the size in bytes of a frame is known by reading its BFS.

8.2.5 Implementations

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 8.1, 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 8.1, 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 was decreasing 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's 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 8.1, is identical to AFOR-2 but employs the *frame stripping* technique. Compared to AFOR-2, the compressed block can contain frames which is then encoded by a single bit frame as depicted in Figure 8.1. 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.

8.2.5.1 Compression and decompression routines

Our implementations rely on highly-optimised routines such as the ones presented in Listing 6.1 and 6.2, 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, with a frame length of 8 values, the routine encodes 8 values using 3 bits each as shown in Listing 6.1, 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. As for 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 ones. This last routine is very fast to execute since there are no shift or mask operations.

8.2.5.2 Bit frame encoding

As a reminder 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 bit 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 the 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 benchmarks.

8.2.5.3 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.

Chapter 9

SkipBlock: Self-Indexing for Block-Based Inverted Lists

SkipBlock is a model for block-based inverted lists, that can be applied as well on traditional inverted lists. It is based on an adaptation of the Skip List data structure. This work is orthogonal to related work on self-indexing structures, since they might be extended to this model.

In this section, the SkipBlock model is presented with its associated search algorithm. We discuss then the impact of the structure's parameters for both Skip List and SkipBLock. We argue after the different strategies available to search within intervals for both structures. Finally we compare thanks to a cost-based model the differences between the Skip Lists and the SkipBlock.

9.1 The SkipBlock Model

The Skip List model works on a record unit, i.e., a synchronization point has a probability p to appear in an inverted list of n records. The SkipBlock model operates instead on *blocks* of records of a fixed size, in place of the records themselves. Consequently, the probabilistic parameter p is defined with respect to a block unit. A synchronization point is created every $\frac{1}{p^i}$ blocks on a level i, thus every $\frac{|B|}{p^i}$ records where $|B|$ denotes the block size. A synchronization point links to the first record of a block interval. Compared to Figure 7.1, a SkipBlock structure with $p = \frac{1}{8}$ and $|B| = 2$ also has an interval of $\frac{|B|}{p^1} = 16$ records. However, on level 2, the synchronization points are separated by $\frac{|B|}{p^2} = 128$ instead of 256 records. We note that with $|B| = 1$, the SkipBlock model is equivalent to the original Skip Lists model, and therefore it is a generalization. The two properties of the Skip Lists are then translated to this model:

1. the number of levels is defined by

$$L_B(n) = \left\lfloor \ln_{\frac{1}{p}} \left(\frac{n}{|B|} \right) \right\rfloor \tag{9.1}$$

2. the structure's size

$$S_B(n) = \sum_{i=1}^{L_B(n)} \left\lfloor \frac{n \times p^i}{|B|} \right\rfloor \tag{9.2}$$

$\|I\|$	2	16	64	128	1024
$S(n)$	99 999 988	6 666 664	1 587 300	787 400	97 751
C	54	112	320	512	3072

(a) Skip Lists with $|I| = \frac{1}{p}$.

$\|I\|$	16		64		128		1024	
p:$\|B\|$	$\frac{1}{4}$:4	$\frac{1}{8}$:2	$\frac{1}{4}$:16	$\frac{1}{8}$:8	$\frac{1}{4}$:32	$\frac{1}{8}$:16	$\frac{1}{4}$:256	$\frac{1}{8}$:128
$S_B(n)$	8 333 328	7 142 853	2 083 328	1 785 710	1 041 660	892 853	130 203	111 603
C	48	64	44	56	40	56	36	48

(b) SkipBlock with $|I| = \frac{|B|}{p}$.

Table 9.1: Search and size costs of Skip Lists and SkipBlock with $n = 10^8$. $|I|$ stands for an interval length. C reports the search complexity to find an interval (Sections 7.2.2 and 9.1.1).

9.1.1 SkipBlock Search Algorithm

With the block-based Skip Lists model, the search algorithm returns an interval of blocks containing the target record. We discuss the search in that interval in Section 9.2. The search walk is identical to the one presented in Section 7.2.2: we walk from the top to the bottom level, and compare at each step the current synchronization point with the target. The walk stops at the same termination criteria as in Skip Lists. The search complexity in the worst case becomes

$$C_{SB} = \frac{L_B(n)}{p} \tag{9.3}$$

9.1.2 Impact of the Self-Indexing Parameters

In this section, we discuss the consequences of the probabilistic parameter on the Skip Lists data structure. Table 9.1a reports for low (i.e., $\frac{1}{1024}$) and high (i.e., $\frac{1}{2}$) probabilities (1) the complexity (7.3) to find the interval containing the target record, and (2) the size (7.2) of the Skip Lists structure. For example with an interval $|I| = 1024$ and an inverted list of size 10^6 (i.e., Skip Lists with one level), reaching a record located at the end of the inverted list incurs a lot of synchronization points at level 1 being read. With a smaller interval, e.g. 16, the number of synchronization points read decreases because the $L(10^6) = 4$ levels provide more accessing points to the inverted list.

Large intervals give the possibility to skip over a large number of records by reading a few synchronization points. However this leads to less use cases of the Skip Lists structure when the searched records are close to each other. In the latter case, small intervals are then more adapted, but at the cost of a bigger structure. Moreover the more levels there are, the more walks up and down on them there are.

There is a trade-off to achieve when selecting p: a high probability provides a low search complexity but at a larger space cost, and a low probability reduces considerably the required space at the cost of higher search complexity.

The SkipBlock model provides two parameters to control its Skip Lists structure: the probabilistic parameter p and the block size $|B|$. The block size parameter enables more control over the Skip Lists structure. For example, to build a structure with an interval of length 64, the original Skip Lists model proposes only one configuration given by $p = \frac{1}{64}$. For this same interval length, SkipBlock proposes all the configurations that verify the equation $\frac{|B|}{p} = 64$. Table 9.1b reports statistics of some SkipBlock configurations for the same interval lengths as in Table 9.1a. Compared to Skip Lists on a same interval length, SkipBlock shows a lower search complexity in exchange of a larger structure.

9.2 Searching within an Interval

The Skip Lists and SkipBlock techniques enables the retrieval of an interval given a target record. The next step is to find the target record within that interval. A first strategy (S1) is to linearly scan all the records within that interval until the target is found. Its complexity is therefore $O(|I|)$.

SkipBlock takes advantage of the block-based structure of the interval to perform more efficient search strategy. Here are defined four additional strategies for searching a block-based interval, parameterized to a probability p. The second strategy (S2) performs

(a) a linear scan over the blocks of the interval to find the block holding the target and

(b) a linear scan of the records of that block to find the target.

The search complexity is $\frac{1}{p}+|B|$ with $\frac{1}{p}$ denoting the linear scan over the blocks and $|B|$ the linear scan over the records of one block. Similarly to S2, the third strategy (S3) performs the step (a). Then, it uses to find the target an inner-block Skip Lists structure restricted to one level only. The complexity is $\frac{1}{p}+\frac{1}{q}+\lfloor |B| \times q \rfloor$ with q the probability of the inner Skip Lists. In contrast to S3, the fourth strategy (S4) uses a non-restricted inner-block Skip Lists structure. The complexity is $\frac{1}{p}+\frac{L(|B|)+1}{q}$ with q the inner Skip Lists probability. The fifth one (S5) builds a Skip Lists structure on the whole interval instead of on a block. Its complexity is then $\frac{L\left(\frac{|B|}{p}\right)+1}{q}$, with q the inner Skip Lists probability. The strategies S3, S4 and S5 are equivalent to S2 when the block size is too small for creating synchronization points.

9.3 Cost-Based Comparison

In this section, we define a cost model that we use to compare five SkipBlock implementations and the original Skip Lists implementation.

Cost Model For both the Skip Lists and the SkipBlock, we define a cost model by (a) the cost to search for the target, and (b) the cost of the data structure's size. The search cost consists in the number of synchronization points traversed to reach the interval containing the target, plus the number of records scanned in that interval to find the target. The size cost consists in the total number of synchronization points in the data structure, including the additional ones for S3, S4 and S5 in the intervals.

Implementations We define as the baseline implementation, denoted I_1, the Skip Lists model using the strategy (*S1*). We define five implementations of the SkipBlock model, denoted by I_2, I_3, I_4, I_5 and I_6, based on the five interval search strategies, i.e., *S1*, *S2*, *S3*,*S4* and *S5* respectively. The inner Skip Lists in implementations I_4, I_5 and I_6 is configured with probability $q=\frac{1}{16}$. The inner Skip Lists in I_5 and I_6 have at least 2 levels. The size costs of the six implementations are

I_1: $S(n)$

I_2: $S_B(n)$

I_3: $S_B(n)+\frac{n}{|B|}$

I_4: $S_B(n)+\lfloor n \times q \rfloor$

$\|I\|$	8						16						512						1152					
	I_1	I_2	I_3	I_4	I_5	I_6	I_1	I_2	I_3	I_4	I_5	I_6	I_1	I_2	I_3	I_4	I_5	I_6	I_1	I_2	I_3	I_4	I_5	I_6
SC	72	56	54				112	62	56				1536	548	120	64	86	84	3456	1186	154	74	84	81
ZC$\times e6$	14.3	16.7	50.0				6.7	12.5	25.0				0.2	0.3	1.8	6.5	7.0	6.9	0.09	0.17	1.7	6.4	6.6	6.7

Table 9.2: Search (i.e., SC) and size (i.e., ZC, in million) costs with $n = 10^8$. SkipBlock implementations report the best search cost with the associated size cost.

I_5: $S_B(n) + \frac{S(|B|)\times n}{|B|}$

I_6: $S_B(n) + \frac{S(p\times|B|)\times n}{p\times|B|}$

Comparison With respect to the SkipBlock model, we tested all the possible configurations for a given interval length. We report that all of them were providing better search cost than the baseline. We only report in Table 9.2 the configurations providing the best search cost given an interval length with the associated size cost. We observe that I_2 already provides better search cost than the baseline I_1 using the same search strategy *S1*, in exchange of a larger size cost. The other implementations, i.e., I_3, I_4, I_5 and I_6 which use a more efficient interval search strategies further decrease the search cost. In addition, their size cost decreases significantly with the size of the interval. On a large interval (512), I_4 is able to provide a low search cost (64) while sustaining a small size cost ($6.5e6$). The inner Skip Lists of I_5 and I_6 are built on lists too small to provide benefits. To conclude, I_4 seems to provide a good compromise between search cost and size cost with large intervals.

9.3.1 Skipping Analysis

The SkipBlock structure, as a generalization of the Skip List, possesses the same hierarchical layout. However the difference between the two is concrete: SkipBlock jumps over groups of blocks at a time, whereas Skip Lists jumps over groups of records. The Table 9.3 reports the intervals at each levels for both structures, with a constant interval between towers of 256 records. For a same interval length (e.g., 256 records in the table), we are able to yield more configurations with the SkipBlock than with the Skip List model. We remark that in both configurations of the SkipBlock, in-between levels have been added in comparison to the original Skip List. These additional levels give more skipping possibilities while searching. Moreover this implies that more data will be read from a tower in a SkipBlock structure than with the Skip List, for a same interval length.

The SkipBlock structure will behave differently depending on the parameter configurations in comparison to the Skip list, since they yield different skip levels. To have a better understanding of the differences between the two models, we discuss here the cost of reading skip data into memory and in which case it outweighs the CPU saving thanks to an adaptation of the method from [36].

Let t_r be the cost of reading one record or a synchronization point, both as part of a bulk read. Let also t_d be the cost of decoding a record/synchronization point. Let further k be the number of *candidates*, i.e., the records that will be lookup from the inverted list thanks to the self-indexing structure. Then the total time T required to search one inverted list of size n in the worst case scenario (i.e., all levels up to the maximum $L(n)$ are read) is given by the formula

$$T = T_d + T_r = kt_d\left(C + |I|\right) + t_r\left(n + 2 \times S\right)$$

Level	Skip Lists	SkipBlock	
	p=256	B=16 p=16	B=64 p=4
1	256	256	256
2	65 536	4 096	1 024
3	16 777 216	65 536	4 096
4	...	1 048 576	16 384
5	...	16 777 216	65 536

Table 9.3: Interval length for each skip levels.

$\|I\|$	Skip List								SkipBlock							
k	60			6000					60			6000				
	T_d	T_r	T	T_d	T_r	T	SC	ZC	T_d	T_r	T	T_d	T_r	T	SC	ZC
16	0.010	0.034	0.044	0.960	0.034	0.994	64	3998	0.006	0.035	0.041	0.600	0.035	0.635	40	4996
32	0.019	0.032	0.051	1.92	0.032	1.952	128	1934	0.008	0.034	0.042	0.81	0.034	0.844	54	3743
256	0.077	0.030	0.107	7.68	0.030	7.710	512	234	0.041	0.030	0.071	4.08	0.030	4.110	272	309
512	0.154	0.030	0.184	15.36	0.030	15.390	1024	117	0.079	0.030	0.109	7.89	0.030	7.920	526	229
1024	0.307	0.030	0.337	30.72	0.030	30.750	2048	58	0.155	0.030	0.185	15.54	0.030	15.570	1036	75

Table 9.4: Predicted processing times in seconds (n=60,000), with the corresponding structure's search (i.e., SC) and size (i.e., ZC) costs. T_d stands for the time for decoding a record, T_r for the time of reading into memory a record/synchronization point and T the sum of these two times. k indicate the number of candidates.

where T_d is the total time to decode pointers, T_r the total time to read into memory the inverted list and synchronization points. ZC and SC reports respectively the size and the search costs. The numerical values of t_d and t_r are respectively 2.5×10^{-6} and 0.5×10^{-6}, values taken from [36].

The Table 9.4 reports a numerical comparison between the two self-indexing models, with $n = 60,000$ and $k = \{60,\ 6000\}$. With a bigger number of candidates we can expect a bigger processing time, since the self-indexing structure will be accessed more often. The SkipBLock structure's values are reported with the strategy S1 for the configurations that yields the best processing time T. On small intervals (e.g., 16) the SkipBlock model does not provide significant better performance even though the complexity is better than the Skip List. This is explained by the difference in size, as the amount of data to be read is greater, which is reflected by the greater required time to read data T_r. As the interval increases, we can see that the processing times increase for both models. However the expected processing time is inferior with SkipBlock than with the original Skip List by 2 times. The reason is that even if the strategies for searching within intervals is the same (i.e., linear), the additional levels that the SkipBlock model provides give more skipping possibilities. This comforts the fact that the SkipBlock complexities are lower than the Skip List's. With the interval $|I| = 1024$, the structure's size increases by 30%, however both the search complexity and the estimated processing time are twice as low as the original model.

Part IV

Benchmarks

Chapter 10

Benchmarking Framework

The benchmarks were all performed using JAVA, and some precautions should be taken when doing so. Indeed JAVA is a language which is "Just-In-Time" (JIT) compiled, meaning that the code used is compiled only at runtime. JAVA uses different heuristics so that only the most frequently used code is compiled, which optimizations take place while runtime. Thus the time spent by the Java Virtual Machine (JVM) becomes a noise to benchmarks results evaluating the running time of an application. This noise problem should be taken even more seriously when performing micro-benchmarks in order to have sound results to discuss on.

This Chapter presents the different points to pay attention to, before describing the benchmarking platform used to get all the experiments results in this report.

10.1 JAVA and Benchmarking Pitfalls

In this section, some of the important points to take care of in micro-benchmarking with JAVA are presented. The implementation of our benchmark platform is based on the technical advices from [12], where more details about the technical aspects can be found.

Time recording With micro-benchmarking, the running time is generally below 10 ms. Thus using the method #System.currentTimeMillis() is not adapted. An other method for recording such short running times is available within that same class, #System.nanoTime(), which gives a time at the nanosecond precision.

Warm up The JVM works on the code while it is being run. Indeed the JVM is based on just-in-time (JIT) compilation, thus the JVM performs some optimizations on the code at runtime, thanks to various statistics gathered. In order to decrease the noise because of those heuristics, the portion of code being evaluated has to be run several times, without any measures being recorded. In doing so, the classes that are used during the benchmark are already "loaded" by the JVM. [12] reports that doing so for 10 seconds is enough to have most of the JVM optimizations done.

Dynamic optimization Even though the warm up phase decreases the number of optimizations likely to be done by the JVM hereafter, it is still possible that some are executed while measuring the running time. In order to detect these changing state of the JVM, we can use the JAVA classes *ClassLoadingMXBean* and *CompilationMXBean* within the package "java.lang.management" that reports various statistics about the JVM, e.g. the number of loaded classes.

On line 2 and 3, we instantiate objects that gives information about the state of the JVM. ClassLoadingMXBean reports the number of classes that have been loaded/unloaded since the JVM started. CompilationMXBean reports the time ins ms spent on compilation. By wrapping the evaluated code with such recordings of the JVM (lines 5 to 7 and 13 to 15), we are able to detect that the JVM performed an optimization by testing those variables: if they are different, the JVM state changed and so the evaluation need to be restarted. This allows to make sure that the evaluated time returned is as "clean" as possible.

```
 1  // MF is the namespace used for ManagementFactory
 2  ClassLoadingMXBean loadBean = MF.getClassLoadingMXBean();
 3  CompilationMXBean compBean = MF.getCompilationMXBean();
 4
 5  long loadedClasses1 = loadBean.getTotalLoadedClassCount();
 6  long unloadedClasses1 = loadBean.getUnloadedClassCount();
 7  long compiledTime1 = compBean.getTotalCompilationTime();
 8
 9  long t1 = System.nanoTime();
10  // Evaluated Code
11  long t2 = System.nanoTime();
12
13  long loadedClasses2 = loadBean.getTotalLoadedClassCount();
14  long unloadedClasses1 = loadBean.getUnloadedClassCount();
15  long compiledTime2 = compBean.getTotalCompilationTime();
```

Memory management The memory usage of an application is managed by the JVM through the use of two mechanisms that are automatically executed. The user as no control over them and can occur any time the JVM deems necessary.

1. **Garbage Collection:** free the allocated memory by objects that are no more used. It can be explicitly called by *#System.gc()*.
2. **Object Finalization:** any objects possess the #finalize() method, which frees all resources used by this object. This method is called by the garbage collection mechanism. However it is possible to explicitly run this method for all objects which destruction is pending by calling the method *#System.runFinalization()*.

The time spent on running those two mechanisms is an other source of noise in the benchmark. This can be prevented to some point by calling the #System.gc() and #System.runFinalization() methods until the memory usage stabilizes.

The loop from line 11 to 24 tries to free by force with calls to the previous presented methods. The loop is repeated until the consumed memory stabilizes and that the number of objects that are waiting to be destroyed are no more.

```
1  // Return the amount of memory currently used
2  long usedMem() {
3     final Runtime rt = Runtime.getRuntime();
4     return rt.totalMemory() − rt.freeMemory();
5  }
```

```

final MemoryMXBean mem = ManagementFactory.getMemoryMXBean();
long usedMemPrev = usedMem();

// Upper bound of 100 iterations in order to prevent an ininite loop
for (int i = 0; i < 100; i++) {
   // Trying to free memory
   System.runFinalization();
   System.gc();

   final long usedMem Now = usedMem();
   // Return the number of objects waiting for finalization
   final int county = mem.getObjectPendingFinalizationCount();

   if ((oCount == 0) && (usedMemNow >= usedMemPrev)) // Memory usage stable
      break;
   else
      usedMemPrev = usedMemNow;
}
```

10.2 Experimental Environment

In this section is presented the benchmarking framework that is used for all benchmarks results presented in the next chapters.

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[1].

Experimental Design The benchmarking design below takes into consideration the different critical points that were previously presented. Each evaluation of the benchmark where made by

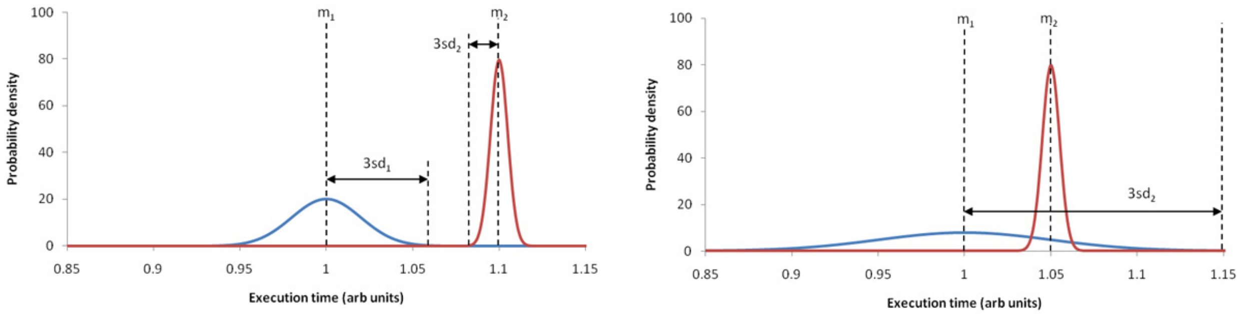
Initialisation of the benchmark environment

1. flushing the OS cache;
2. initialising a new JVM;
3. warming the JVM by executing a certain number of times the benchmark until 10 seconds is reached.

Evaluation of the benchmarked code

4. Each measurement is made by performing n times the evaluated code execution, 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%. The measurement time is the CPU time, i.e., user time and system time, used by the current thread.

[1] Apache Lucene: http://lucene.apache.org/



(a) Non-overlapping standard deviations.

(b) Overlapping standard deviations.

Figure 10.1: Benchmarks results of two measurements reported with means (m_i) and standard deviations (sd_i).

5. the previous step is re-run if the JVM state changed.
6. the average running time is recorded.
7. steps from 4 to 6 are repeated 100 times.

   As recommended in [31], we report the harmonic mean and the standard deviation of the 100 measurements.

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.

10.2.1 Measurements

The benchmark produces 100 measurements for one evaluation so that an average running time can be given, with its standard deviation. However one has to be wary regarding such results. Indeed while the standard deviation allows to have a better critic about the running time, it can also lead to misinterpretations.

The *Vysochanskï-Petunin* inequality shows that 95% of the measurements lie within an interval of three times the standard deviation around the mean. When comparing two different evaluations with these statistics, one cannot say which one is faster if that interval around the mean overlaps. The Figure 10.1a depicts the case where two evaluations have their intervals non-overlapping, in which case it is possible to say that the evaluation with the lower mean is better. However this is not the case if intervals do overlap, as it is the case in the Figure 10.1b. Indeed we can have a measurement belonging to evaluation 2 faster than one from the evaluation 1, even though the mean of evaluation 1 is lower than the mean of evaluation 2. In such a case, we cannot say which one is faster, only that the two evaluated codes are similar.

Chapter 11

Compression Benchmarks

This section describes the benchmark experiments which aim to compare the various compression methods described previously. The benefits gained with the use of AFOR algorithms is discussed through two types of data sources. The first one are unstructured documents: a document is just a bag of words. The second one are semantically structured documents, e.g. RDF enriched documents. Since the indexing model for these two types are different, the distribution of the values within the streams is then also different. Thus the efficiency of a same compression technique depends on the dataset type.

11.1 Benchmarks Settings

This section describes the benchmark experiments which aim to compare the techniques introduced by AFOR with the compression methods described in Section **??**. The first experiment measures the indexing performance based on two aspects: (1) the indexing time; and (2) the index size. The indexing time corresponds to the time required to build the index, be it a traditional inverted index or a node-based inverted index like SIREn. As for the index size, this time will vary with the compression technique used. The second experiment compares the query execution performance using indexes built with different compression techniques.

Data Collection In these benchmarks, five datasets were used. Two of them are plain text datasets which will be used to build traditional inverted indexes, while the three others are RDF triples and so will be used to build SIREn indexes.

$\Longrightarrow$ Datasets for traditional indexes are composed of:

Wikipedia: set of English language Wikipedia articles (2.5 million articles). Its total size is 42GB uncompressed.

Blog: set of 44 million blog posts made between August 1st and October 1st, 2008 [13]. Its total size is 142GB uncompressed.

$\Longrightarrow$ Datasets for the SIREn structure are the following three:

Geonames: a geographical database and contains 13.8 million of entities[1]. The size is 1.8GB compressed.

[1] Geonames: http://www.geonames.org/

DBPedia: a semi-structured version of Wikipedia and contains 17.7 million of entities[2]. 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 5.1.

11.2 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 and finally by optimising the index.

We report the results of the indexing experiments in Table A.1. 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 A.1 is studied based on the size of each values' streams in the inverted file, and on the total size of the index. For indexes built with unstructured data (i.e. Wikipedia and Blog), these streams represent the document identifier, the frequency and the positions. As for the indexes built on structured data (i.e. DBpedia, Geonames, Sindice), the streams represent the entity, frequency, attribute, value and position values. In each tables, the total size is computed by summing the size of each streams. Bar plots are also provided in order to visualise better the differences between the techniques.

Commit Time Figure 11.1 shows the total time spent by each method. As might be expected, Rice is the slowest method due to its execution flow complexity on both source type.

On the large unstructured dataset Blog, PFOR is equivalent to Rice in term of compression speed, since PFOR spends more time in finding the outliers. AFOR-2 and S-64 perform similarly, while AFOR-1 performs as fast as VByte. The drop of efficiency between AFOR-2 and AFOR-1 is due to the optimization algorithm to find the best compressing frame. On the small dataset Wikipedia, Rice is followed by S-64. While FOR, PFOR and VByte perform similarly, AFOR-1 and AFOR-2 algorithms are the most efficient methods.

On small structured datasets (DBpedia and Geonames), algorithms behave similarly as described previously. AFOR-3 shows equivalent performance as S-64 on DBpedia, while performing as the best one on Geonames. 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 large datasets (Sindice and Blog), VByte provides the best commit times, reflecting the benefit from a simple algorithm.

Optimisation Time Figure 11.2 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.

On both data sources types, while PFOR was similar to Rice in term of commit time on large datasets (Blog and Sindice), it is ahead of Rice in term of optimisation time. This can be seen too with FOR on

[2] DBpedia: http://dbpedia.org/

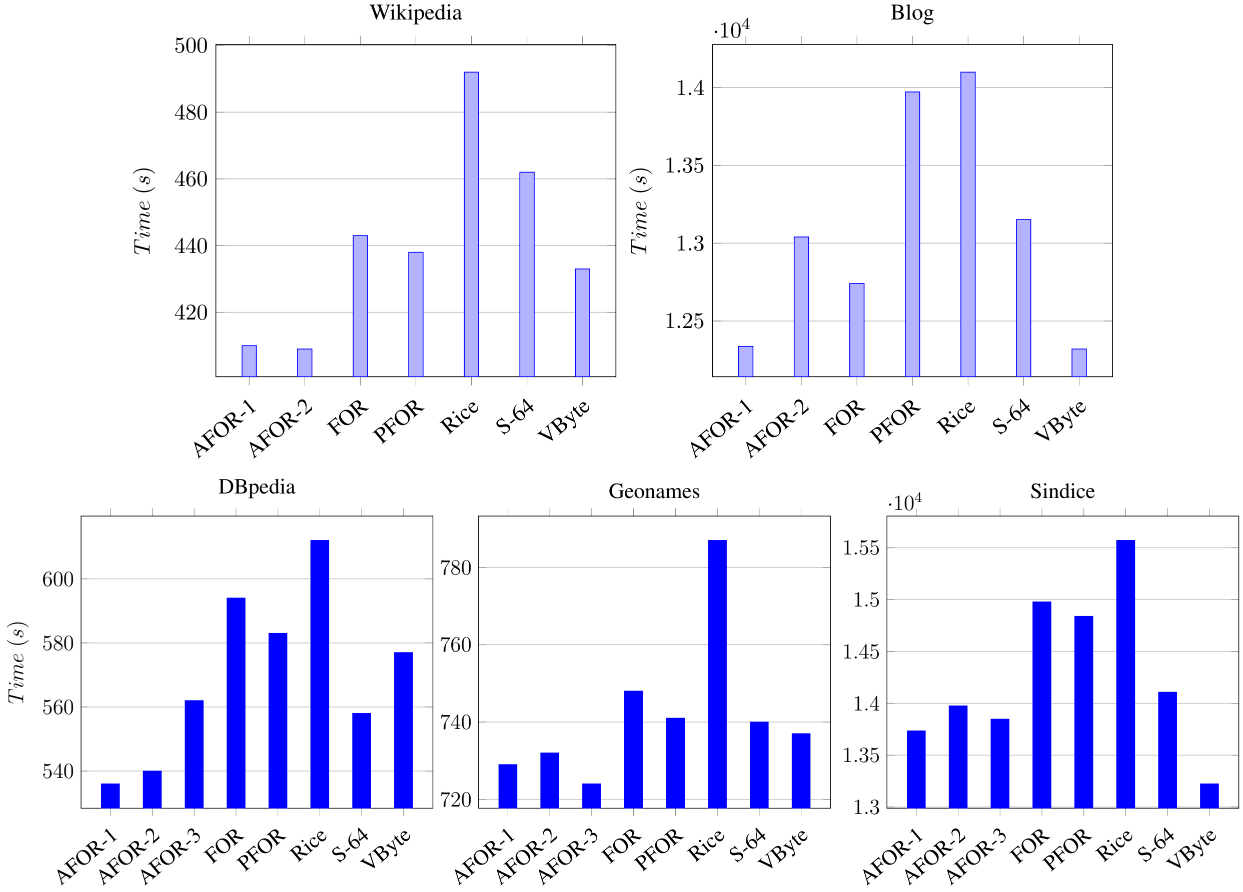


Figure 11.1: The total time spent to commit all the batches of 10000 document.

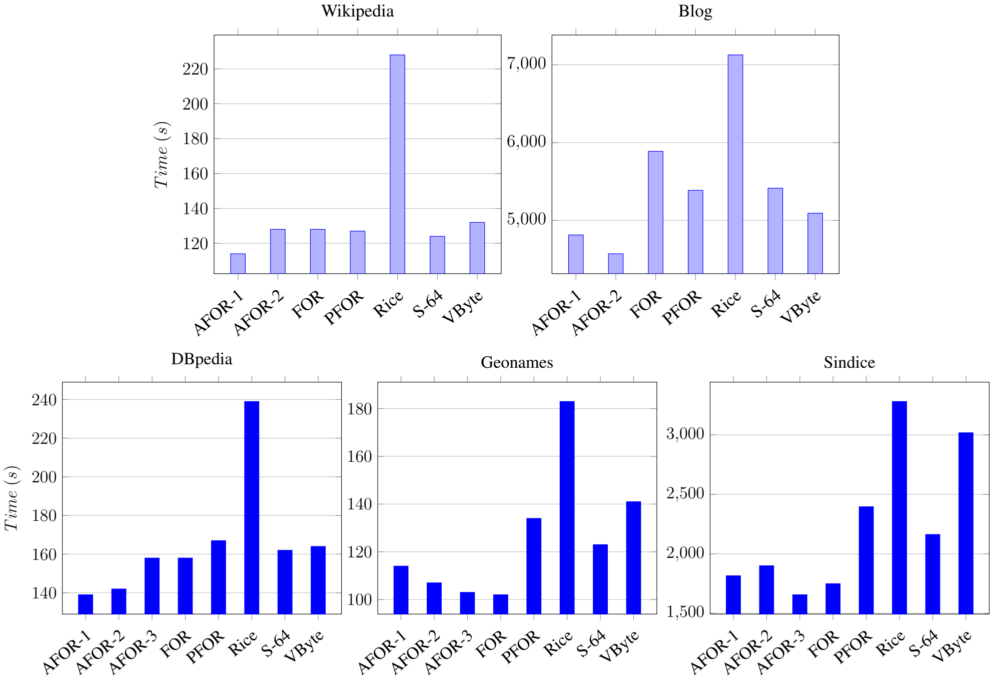


Figure 11.2: The total time spent to optimise the complete index.

Sindice only. Rice is penalised by its low decompression speed. The difference introduces by the decompression performance can also be seen with S-6, which provides performance close to or even better than VByte. While VByte performs fairly well on unstructured data, we notice a large overhead on Sindice. The reason is the distribution of the values on each stream: SIREn produce small values, since there is a locality effect, i.e. attributes are local to an entity, values are local to an attribute and positions are local to a value. On traditional inverted index, position values are local to the whole document, and are then much bigger. The Table A.1 reports a positions stream of size 20 GBytes at least with every compression techniques, while the SIREn's streams sizes are bellow 2 GBytes. This difference in compression ratio produce a large overhead on large datasets (Sindice) with the decompression speed of VByte.

The best-performing methods are AFOR-1, AFOR-2 and AFOR-3 (structured data only), 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 11.3 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 like with Wikipedia and Blog datasets, this is not true for inverted indexes based on a node indexing scheme like SIREn. 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 to encode 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. Indeed we can see that the distribution of values with

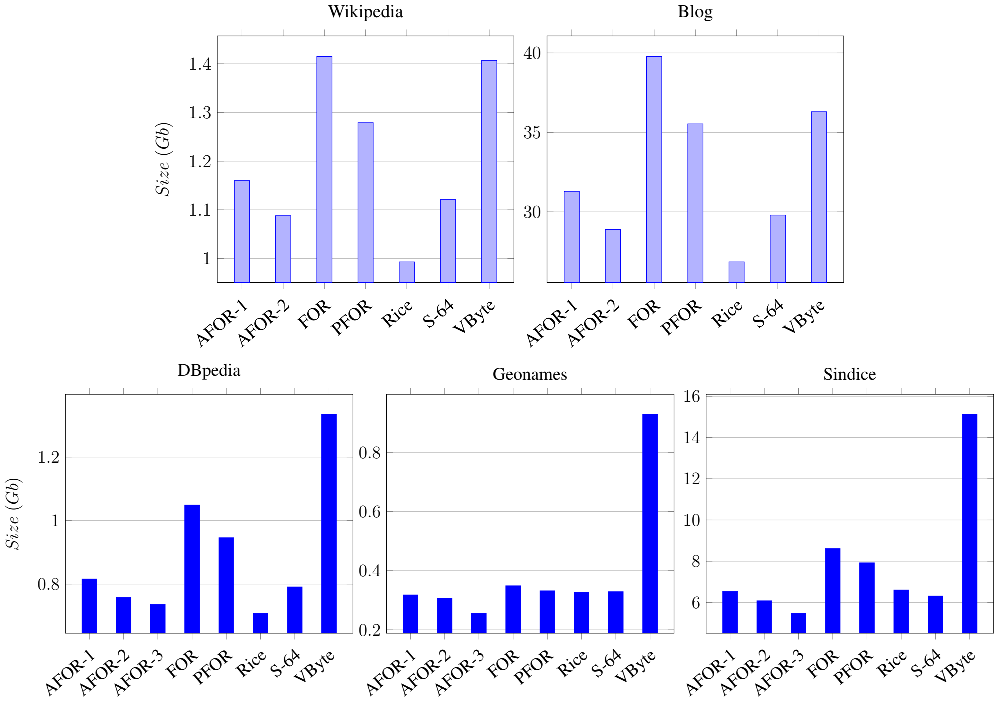


Figure 11.3: The index size achieved by each compression technique.

a document-centric indexing technique (Wikipedia and Blog) lead to many outliers, and thus the PFOR algorithm is in that case efficient.

In contrast, AFOR and S-64 are able to better adapt the encoding to the value distribution and therefore provide a better compression rate. While Rice provides the best compression ratio on traditional inverted indexes, AFOR is able to provide better compression ratio than Rice on the Geonames and Sindice dataset. Compared to AFOR-2, we can observe in Table A.1 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 8.2.3.

In comparison to normal indexes, Rice provides worse compression ratio than AFOR on a node indexing scheme index. This can be explained again by the small values, since Rice does not compress long runs of small values as well as AFOR.

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 the AFOR family, and this on both structured and unstructured datasets. 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.

11.3 Querying Performance

We now compare the decompression performance in real settings, where inverted indexes are answering queries of various complexities. We report in this section only the benchmarks done on structured datasets (DBpedia, Geonames and Sindice), since the performance of query processing on unstructured datasets is comparable to the ones run on the structured datasets and for question of clarity also. The raw results are nonetheless reported in the appendix in the Table A.3.

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.

11.3.1 Query Benchmarking Framework

In this section we present the types of queries that are run against indexes compressed with different algorithms.

11.3.2 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 [21] to obtain 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[3] from the data collection. We then compute their frequency and sort them by descending frequency. We finally create the three ranges as explained above. 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 one presented here.

11.3.2.1 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, e.g. to find all entities which have the word “fantasy” within a value as in 5.1. Therefore, the processing of conjunction and disjunction queries relies on the entity, frequency, attribute and value inverted files. Phrase queries rely on one additional stream, the position values.

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

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

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.

11.3.2.2 Attribute Queries

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

11.3.3 Query Benchmark Design

The benchmarking design used is the one presented in the Chapter 10. 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. All measurements are made using *warm cache*, i.e., the part of the index read during query processing is fully loaded in memory.

Query execution time is sensitive to external events which can affect the final execution time recorded. To assess differences between the algorithms, confidence intervals with 95% degree of confidence have been used. 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.

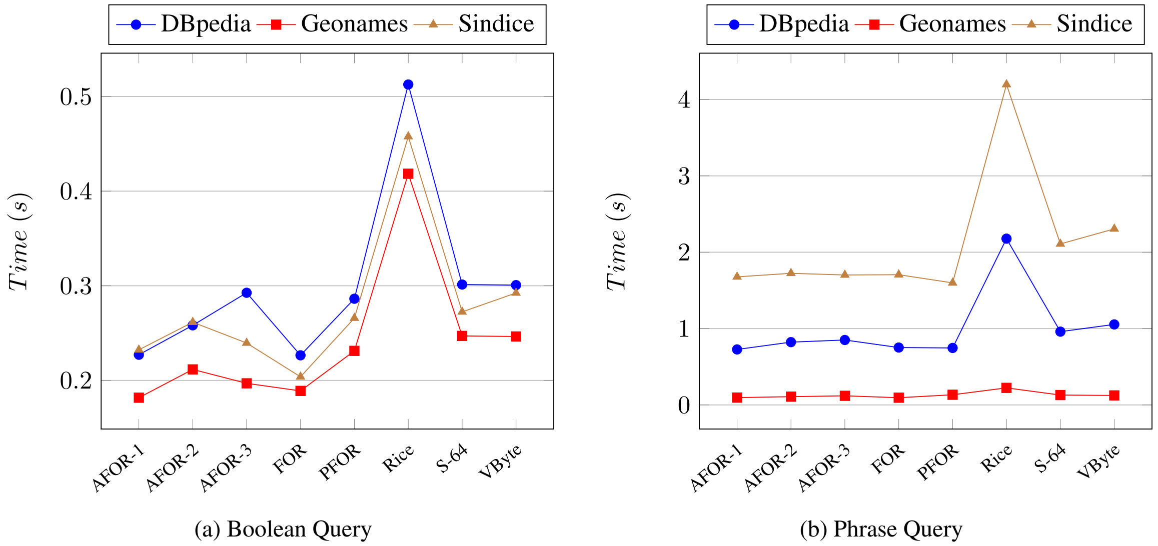
Each condition of the design, e.g., AFOR-1 / 2-AND / DBpedia, contains 100 separate measurements.

11.3.4 Query Benchmark Results

We report the results of the query benchmarks in Table A.2a and Table A.2b in the appendix 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 11.4 and Figure 11.5 report the average query processing time for the value and attribute queries respectively. Figure 11.4a and Figure 11.5a depict the average processing time on the Boolean queries. Figure 11.4b and Figure 11.5b depict the average processing time on the phrase queries (2-Phrase, 3-Phrase). The query processing time are obtained by summing up the average time of each query from Table A.2a for the value queries and Table A.2b for the attribute queries. For example, the processing time of AFOR-1 on the DBpedia dataset in Figure 11.4b 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 A.2a in the appendix.

Value Query In Figure 11.4, 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.



(a) Boolean Query

(b) Phrase Query

Figure 11.4: The average processing time for the value queries that is achieved by each compression technique.

Rice has the worst performance for every query and dataset, followed by VByte. Nonetheless Rice performs in many cases twice as slow as VByte. In Figure 11.4a, S-64 provides similar performance to VByte on Boolean queries but we can see in Figure 11.4b 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.

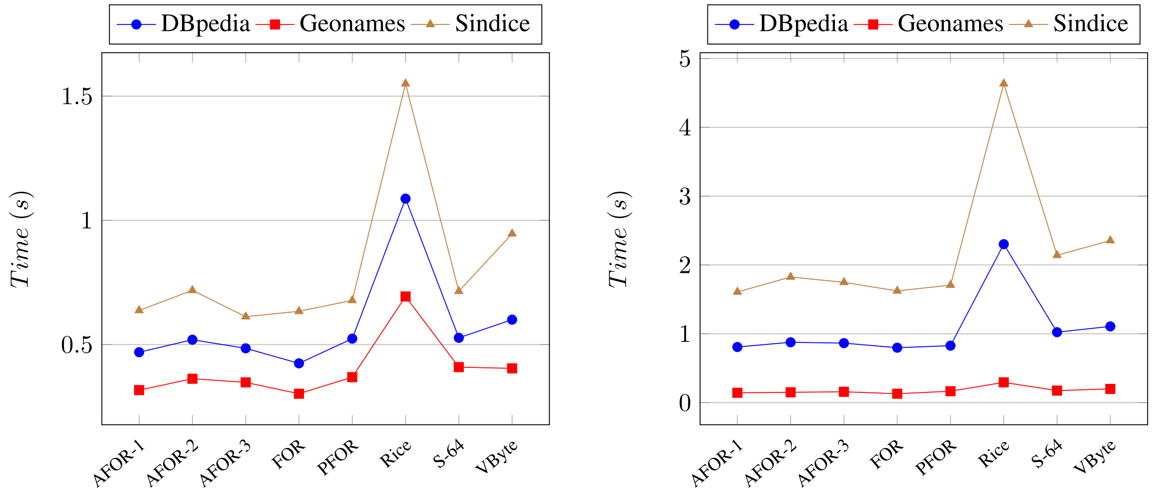
Attribute Query In Figure 11.5, and in particular on Sindice, we can again distinguish the same three classes of algorithms. However, the performance gap between S-64 and VByte becomes wider.

Rice has again the worst performance for every query and dataset. Compared to the performance on value queries, we can see in Figure 11.5a 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 11.5b, 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.

11.4 Performance Trade-Off

We report in Figure 11.6 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



(a) Boolean Query

(b) Phrase Query

Figure 11.5: The average processing time for the attribute queries that is achieved by each compression technique.

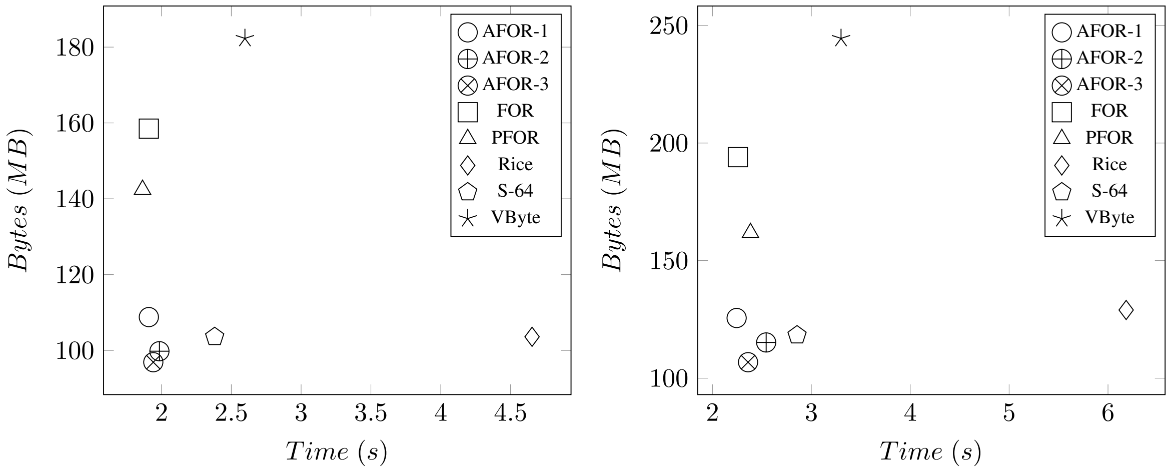
average time of all the queries. The compression ratio is based on the number of bytes read during query processing which are reported in Table A.2a and Table A.2b in the appendix. We can distinctively see that the AFOR techniques are close to Rice in term of compression ratio, while being relatively close to FOR and PFOR in term 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 processing time close to AFOR-1.

We report in Figure 11.7 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 A.1 of the appendix. 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.

11.5 Discussion

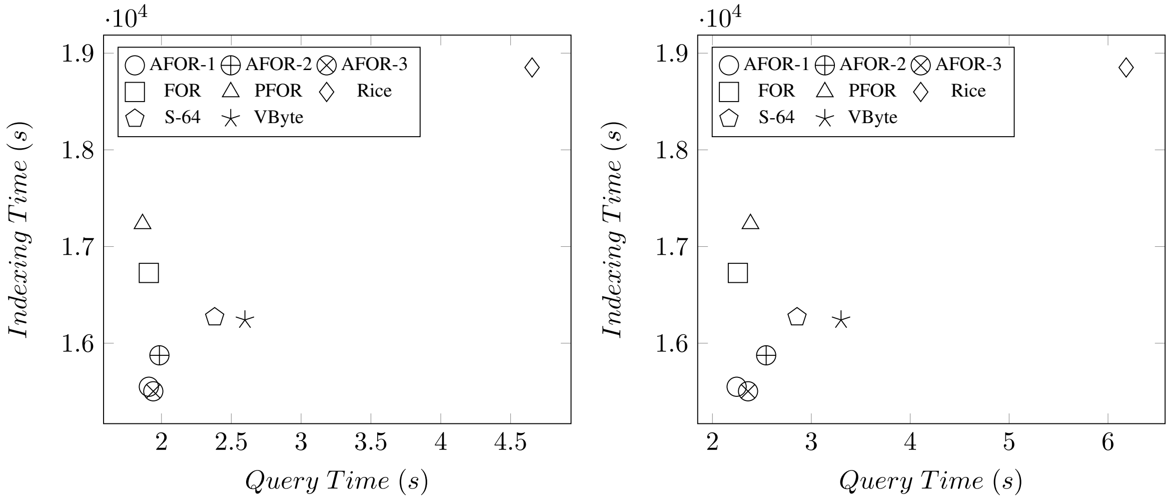
In general, even if FOR has more data to read and decompress, it still provides one of the best query execution time. The reason is that our experiments are performed using warm cache. We therefore ignore the cost of disk IO accesses and measure exclusively the decompression efficiency of the methods.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

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



(a) Value Query (b) Attribute Query

Figure 11.6: 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 11.7: A graphical comparison of the compression techniques showing the trade-off between querying time and indexing time on the Sindice dataset.

are most likely to increase the update and query throughput 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, and is thus simpler to implement and provides better compression ratio, compression speed and decompression speed.

Chapter 12

Scalability of SIREn

In the previous experiments, we have seen that AFOR-3 is the most suitable compression technique for an entity inverted index like SIREn. Based on these results, we perform a stress test of the entity index compressed with AFOR-3 through 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) the system can answer with respect to the size of the index and the complexity of the query.

12.1 Benchmarking Framework

In this section we discuss the ability of the AFOR compression techniques family to scale with the growing size of the compressed data.

Data Collection The full Sindice data collection is currently composed of more than 120 millions of documents among 90.000 datasets. For each dataset, we extracted the entities as depicted in Figure 5.1. We filtered out all the entity descriptions containing less than two values. After filtering, there is a total of 907.542.436 entities for 4.689.599.183 RDF triples. We create three datasets: *Small* containing 226.129.319 entities for 1.240.674.545 RDF triples; *Medium* containing 447.305.647 entities for 2.535.658.099 RDF triples; 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. Using an uniform distribution for the random method, each attribute query is generated by selecting at random an attribute term from the high, medium or low selectivity ranges. The associated value query is generated by selecting at random 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. 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 experiments. 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.

Following the benchmark design in Chapter 10, for each type of star query, we (1) generate a set of 400 random queries, and (2) perform 100 measurements. Each measurement is made 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.

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.

12.2 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 further presented in Chapter **??**, is 7.644 GB. The total size of the index is 35.731 GB which represents an average of 8 bytes per RDF statement (i.e. triple).

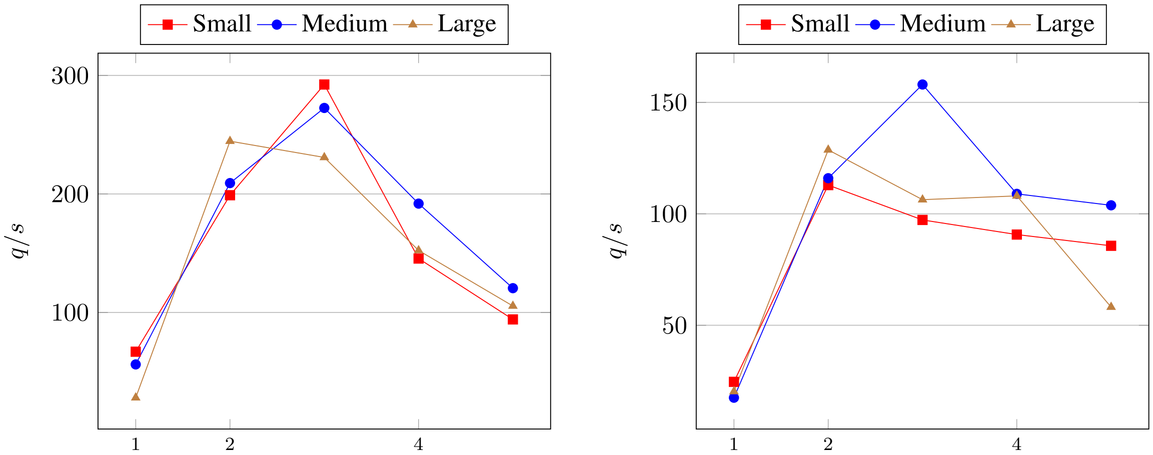
12.3 Querying Performance

We report the results of the scalability benchmark in Table A.4 of the appendix. Based on these results, we derive two graphical charts in Figure 12.1 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. On the contrary, the precision of the query increases with the number of attribute queries, and the chance of having a large number of hits decreases consequently.

Concerning the term selectivity, we can note a drop of query rate between Figure 12.1a where query terms of low selectivity are employed and Figure 12.1b where query terms of low selectivity is not employed. In the later case, the system has to perform more record comparisons.

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,



(a) Query rate with LMH selectivity

(b) Query rate with MH selectivity

Figure 12.1: The evolution of the average query rate with respect to the size of the star queries over different dataset size.

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.

12.4 Discussion

While AFOR provides better compression ratio than PFOR, AFOR-2 is slower than PFOR on position file, suggesting that PFOR exception management is slightly more efficient than the variable frame length approach of AFOR-2 on very sparse lists. The number of table lookups in AFOR-2 costs more than the decoding and patching of the exceptions in PFOR.

In general, FOR even if it has more data to read and decompress still provides 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 so we can expect a drop of performance for algorithms with a low compression ratio such as FOR and PFOR compared to AFOR when executing with a cold cache.

Chapter 13

Self-Indexing Benchmarks

In this section we will discuss the performance of SkipBlock compared to Skip Lists by building structures on real data and performing set operations. First we present the benchmarking environment used for comparing the self-indexing structures. Then we compare the results of the SkipBlock structure to the theoretical ones, before discussing the benefits of the block-based self-indexing structure against the original Skip List.

13.1 Benchmarking Framework

The benchmarking framework is the same as in Chapter 11. In this section I present the benchmark environment used to evaluate the self-indexing structures.

13.1.1 Dataset

For this benchmark we use inverted lists extracted from the Sindice dataset from Section 11.1. Those inverted lists only contains the entity identifiers values, the self-indexing structures being built upon an ordered list of records, and since only the raw performance of the self-indexing structure is measured. The other identifiers (e.g., attributes or values identifiers) are only relevant when computing queries, thus they are discarded for this benchmark. A small set from each frequency groups, i.e., HIGH, Medium and LOW groups generation presented in Section 11.3.1, is extracted but keeping the entities identifiers stream only. A list from the HIGH group will be longer than one from the LOW group.

13.1.2 Benchmark Design

In order to have a better view of the raw performance of the self-indexing structures, we perform two set operations, *exclusion* and *conjunction*. The lists operated on are taken from one of the three frequency groups of the Sindice dataset. Conjunction operations perform the intersection of n lists given as operands. Exclusion operations $L_a \backslash L_b$ exclude all the records from L_b in L_a.

This design is based on the querying benchmark's design from the Section 11.3.1. In order to process conjunctive queries, the inverted lists from each term of the query are retrieved and intersected as explained in Chapter 3. To process the Boolean operator NOT, the inverted list of the term the operator is affected to is excluded from the results. At the core of queries processing, set operation are performed on the retrieved lists. Based on the notation of the Section 11.3.1, the conjunction set operation of two inverted lists reflects the same underlying process when computing 2-AND queries. A measurement of the evaluation consists in the average time to execute a set operation.

The benchmark records four information:

1. the number of bytes read from the self-indexing structure.
2. the number of documents identifiers skipped.
3. the average time needed to perform 1 measure.
4. the total number of search operations done on the structure as defined in Section 9.3, composed of the number of synchronization points read plus the number of records scanned.

The design of the SkipBlock benchmark includes two factors:

Operands having two levels: HIGH-HIGH and HIGH-LOW. For instance, the former value means that there are two operands, the HIGH and LOW inverted lists.

Operation having two levels: exclusion and conjunction.

Each condition of the design, i.e., HIGH-HIGH, Conjunction, possesses 100 distinct measurements.

13.1.3 Implementations

The SkipBlock model possess two implementations based on the interval search strategies introduced in Section 9.2. However only the S1 and S2 strategies will be discussed in these results due to a lack of time to evaluate the others.

- The baseline I_1 is the original Skip Lists structure with the linear search strategy *S1*.
- The implementations I_2 and I_3 use respectively the strategies *S1* and *S2*. As a remainder, the S2 strategy uses block headers as additional synchronization points within an interval.

13.2 Results

For these benchmarks, both the self-indexing structures and the inverted list are compressed with the VByte algorithm. Before experimenting on the set operations, we evaluated the performance of the self-indexing structures solely on skipping records. We first review the evaluation results on the raw performance of the structures, before discussing the performance with set operations.

13.2.1 Advancing on a List

In this section we evaluate the raw performance of both self-indexing models to advance on an ordered list. The Table 13.1 reports these results on a list of 6×10^8 records, records that only consist of entity identifiers. The first two rows reports the size in MBytes of the self-indexing structures. A sequence of equally spaced candidates are searched, i.e., a small skipping length of 16 and a large one of 13×10^4 records. For each skipping length, the self-indexing structure are parameterized with intervals of 32 and 1024, yielding for SkipBlock respectively two possible configurations ($|B| = 8,\ p = 4$) and ($|B| = 64,\ p = 16$).

	$\lvert I\rvert$	I_1			I_2			I_3		
	32	40.37 MB			83.09 MB			297.57 MB		
	1024	2.24 MB			2.57 MB			31.6 MB		
Length		MB	Time	Ops	MB	Time	Ops	MB	Time	Ops
16	32	38.11	12.9 s ± 135.1 ms	338 104 836	59.72	13.8 s ± 96.8 ms	324 999 973	131.25	19.3 s ± 132.1 ms	212 499 973
	1024	2.24	12.6 s ± 117.7 ms	591 797 454	2.46	12.0 s ± 208.6 ms	591 250 005	29.28	14.0 s ± 88.7 ms	459 999 997
130 000	32	0.54	48.0 ms ± 5.8 ms	211 963	0.30	42.3 ms ± 8.8 ms	109 174	0.31	47.8 ms ± 3.1 ms	95 326
	1024	2.10	112.7 ms ± 3.5 ms	2 883 453	0.33	46.1 ms ± 2.8 ms	2 398 582	0.44	39.5 ms ± 2.1 ms	283 796

Table 13.1: Self-indexing structures performance with equally spaced (i.e., $Length$) candidates. *MB* stands for the number of MBytes read from the structure. Ops reports the total number of search operations performed (i.e., the number of synchronization points read plus the number of scanned records).

For a same interval $|I|$ the SkipBlock structures performs less search operations, thus saving CPU cycles as the Table 9.2 have shown this aspect. However this benefit is outweighted by the structure's size on small intervals (i.e., $|I| = 32$), since more data has to be read into memory. Despite the predicted processing times reported in the Table 9.4, the actual processing time for the SkipBlock structure is not what was expected, i.e., half the time of the original Skip List's. This can be explained by the compression algorithm, VByte, which is not suited for compressing blocks.

We can note that for large skip lengths (i.e., 130 000) the SkipBlock structures are more than 2 times as fast as the original Skip List, on large intervals. The reason is that for so large intervals the baseline has its number of levels considerably reduced, which is not the case for SkipBlock since it adds in-between levels (e.g. Table 9.3). On top of these additional levels, I_3 allow to reduce the number of search operations by 10 times in comparison to the baseline. When performing small skips, I_2 and I_3 implementations provide better time than I_1 on large interval.

This experiment showed that the SkipBlock model provides important benefits when jumping over a large number. With small skipping lengths, more data is read from the SkipBlock as there are additional levels. In the latter case the benefit of additional skipping levels is outweighted by the increased amount of read data. We can conclude that in order to get the most benefit from SkipBlock, configurations with large interval lengths are the most suited.

13.2.2 Set Operations Results

For these benchmarks, both the self-indexing structures and the inverted list are compressed with the VByte algorithm, so that the differences in performance between the two structures are not caused by the compression technique.

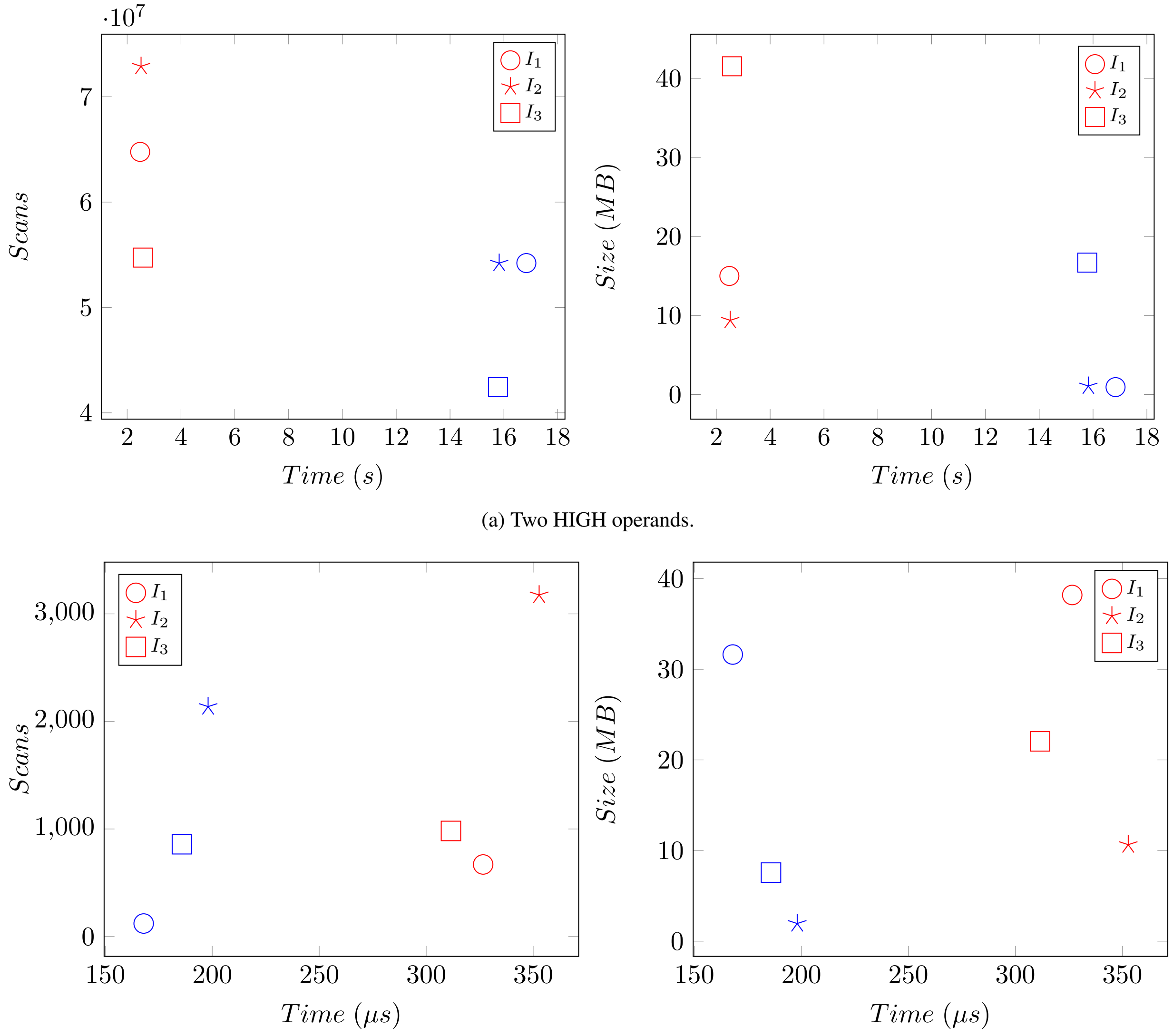
The performance of the self-indexing structures is compared based on the time to perform a set operation, on the number of records that had to be scanned from the inverted list to search for the candidates, and on the size of the structure. We can note that the The raw results are reported in the Table A.5 and Table A.6 in the appendix. For each operands type (i.e., HIGH:HIGH or HIGH:LOW) and implementations, the best execution time is taken and reported in the plots of the Figure 13.1.

We observe from the plots that for either operations on two large lists (i.e., HIGH:HIGH type), the SkipBlock configuration of the implementation I_3 provides slightly better time than the original while reducing the number of scanned records by 10^7. However this has an impact on the structure's size with an increase of 20 MBytes in comparison to the original model. With set operations over lists of different sizes (i.e., LOW:HIGH type), we observe the opposite: by scanning slightly more records and with comparable running times, the implementation I_3 halves the structure's size by at least two.

For set operations on very dense lists, we are able to skip over a large number of records with the SkipBlock model by increasing the structure's size, while still providing similar running time as the original model. For set operations on lists presenting a high size discrepancy, the SkipBlock model is able to greatly reduce the structure's size while still providing similar running times.

With regards to the raw results of the Appendix A.3, this comforts the conclusion of the previous section that the SkipBlock model provides more benefits with large intervals. We can note that the implemented search strategies on intervals are simple ones. Thus it is possible to improve performances by either changing the strategy, or by using an other compression method. Indeed we used for this benchmark the VByte algorithm which is not a block-based algorithm and thus does not profit from all the advantages of the SkipBlock model. With a compression technique such as AFOR, we are effectively able to reduce the size of the structure by using the frame skipping technique (Section 8.2.4).

We can conclude that the SkipBlock model provides implementations, on large interval lengths (e.g., 4096), that trade the structure's size cost over its searching cost.



(a) Two HIGH operands.

(b) Two operands, one HIGH and one LOW.

Figure 13.1: Conjunction (in red) and exclusion (in blue) set operations. The time on the x axis is the best average time, from the Appendix A.3, to perform the operation on the given operands for an implementation.

Part V

Additional Work and Conclusion

Chapter 14

SIREn Query Parser

The query parser is used in SIREn in order to allow people to interact with its index. However we must keep in mind that Sindice, which uses SIREn for searching and browsing purpose, is aimed to be used as a web service by machine as well. In this section, we present a query parser that matches triple patterns in datasets before describing a parser that matches entities.

14.1 NTriple Query Parser

As with the SPARQL language, the NTriple query parser aims to find sub-graph within datasets that best match a given set of triple patterns. Triple patterns are linked together thanks to Boolean operators, i.e. AND, OR, NOT. To express such patterns, SIREn uses a bottom-up grammar, and is built as a plugin to the Lucene[1] query parser.

Literals any string enclosed between simple quotes are taken as a literal.

Literal Pattern any string enclosed between double quotes. It describes the pattern that has to appear within the object of the returned triples for the query, thanks to Boolean operations.

URIs any string enclosed between the characters '<'   '>'.

Any part of a triple pattern, to a maximum of two per triple, can be left unknown thanks to the wildcard character '*'. Any term of the query can be flagged in order to indicate if the term has to, must not or may appear within the triples in the results by using respectively the characters '+', '-', '' (i.e no character). Considering URIs are tokenism when indexing (e.g. http://sindice.com/ is tokenism into "HTTP", "Sindice" and "com"), the following query

<rings> <author> "j.r.r._tolkien OR tolkien" .

matches the RDF graph of the Figure 4.2b. The object of this triple pattern is a literal pattern, which aims to widen the search to several writings of the author's name. Since SIREn is an Information Retrieval search engine, it processes the query as explained in Chapter **??** with set theory operations (i.e. conjunction, disjunction and exclusion).

During indexing, it is possible to store into a specific field any text. For instance, one can have three different fields such as "title", "author" and "abstract" for the entities in the Figure 5.1. This way the abstract

[1] Lucene: http://lucene.apache.org/

of the entity *_:bnode1* would fall into its field, allowing for more precise search. A field is identified with the special Lucene character semi-colon, ':'. For instance the term "hobbit" within the entity _:bnode1 can be refer ed as *abstract:hobbit*.

14.1.1 Added Features

The SIREn query parser has been extended to support commonly used feature in RDF and also to be more expressive. In this section three added features are presented.

URI Pattern As with literal patterns, this extension allows to use Boolean operators within an URI. It is then possible for example to search for RDF graphs where some URI(s) contains both the terms *lord* and *rings* by entering the query <*lord AND rings*> * * . .

Qualified Name - QName As explained in the introduction to Information Retrieval in Chapter **??**, a QName is a term that is used to abbreviate an URI. Whiting an URI, it is possible to use a QName by appending the term to the semi-colon character. The QName is replaced at query processing time by the correct namespace thanks to a file containing the mapping between a QName and its expansion. This expansion is performed on any string that matches a QName format only within an URI, and in that case leaves the term unchanged if no mapping key has been found. Restricting the substitution to URIs allows to QNames-like strings within literals.

A QName can possess several expansion, e.g. the web site[2] reports that the QName *dbpedia* is mapped to three possible expansion

1. `http://dbpedia.org/resource/`
2. `http://dbpedia.org/dbprop/`
3. `http://dbpedia.org/property/`

In order to deal with multi-valued QNames, all possible expansion replace their QName, linked together with the Boolean operation OR. For instance the query

* <dbpedia:title> * .

is expanded to

* <`http://dbpedia.org/resource/` title OR `http://dbpedia.org/dbprop/`title OR `http://dbpedia.org/property/`title> * .

This expansion is rendered possible thanks to the URI pattern extension of an URI presented previously. Thus it allows to perform more generic search and to abstract from the actual schema definition the RDF graphs possess.

[2] QNames namespace lookup for developers: `http://prefix.cc/`

Document-centric			Tabular format
< s >	< p1 >	< o1 > .	< o1 > < o2 o3 >
< s >	< p2 >	< o2 > < o3 > .	

Table 14.1: The tabular format data representation.

Multi-field search SIREn is based on Lucene, which data representation model uses fields. Fields allows to organize logically the information. For example we can split the documents from the collection according to their dataset, so that all documents from a same dataset are within the same field.

However spitting information into fields implies that we lose a global view of the data. To cope with this drawback, we have implemented an extension to the query parser that search documents across fields, enabling the retrieval of triple patterns regardless of the field it actually occurs. Moreover depending on the reliability or the importance that is given to field, it is possible to affect a *weight* value to that field.

As an example, below is shown a RDF graph with two triples taken from two different datasets: “dataset1” and “dataset2”

dataset1: <http://s> <http://p1> “literal” .
dataset2: <http://s> <http://p2> <http://o2> .

While the query with two triple patterns

(<http://s> * ’literal’ .) AND (<http://s> * <http://o2> .)

matches the previous graph thanks to the implicit link, i.e., the first pattern returns the triple from dataset1 and the second one from dataset2, the query

(<http://s> * ’literal’ .) AND (<http://s> * <http://o1> .)

does not, since the second pattern cannot be found.

14.2 Entity Query Parser

With the semantic web representation model such as RDF, we can describe people, products, countries or any other entities. When searching for information we often want results that are relevant for a specific entity, any other results polluting the information retrieved. As a first step, SIREn is indexing entities with an entity-centric rather than a document-centric model. The second step is to build a parser that handles queries for an entity.

The entity query parser is being developed which goal is to matcher patterns for an entity. For example we will able to formulate the query “Find all the people within DERI that are from France”. To better represent relations between an entity and other entities or objects, we use a one-to-many representation model called the *tabular* format. The Table 14.1 depicts triples in a document-centric representation on the left, and its tabular view on the right. With the tabular format we define as a field the predicate (i.e., attribute) and store within that field all the related objects. To answer the previous example query, we can have two fields in the entity description of DERI, the countries and the people fields.

For such queries to be efficient the schema of the data has to be known. As such this parser is intended to be use over specific datasets, such as the intranet of a company.

Chapter 15

Conclusion

In the recent years, the amount of semantic data has considerable increased. More and more web sites are exporting their data using RDF, as the power of the semantic web is increasingly attractive: to be able to efficiently search across multiple data sources specific information, and to retrieve relevant results hereafter. The semantic Web is a challenging and expanding research domain. The Web as we know it is undergoing a radical change, and the semantic web is contributing a lot to it.

People are publishing RDF data following the best practices of Linked Data. The Linking Open Data Cloud is a huge gathering of inter-connected semantic data sources. Applications can use this linked data and provide a concrete benefit to the way we use the Web. *Sig.ma* is a use case for a mashup application that shows the power of the semantic. Its purpose is to search across many data sources, and to provide information organized around *entities*, e.g., a product, a person or any other concept. To be able to efficiently use this collection of knowledge of we scale so that applications like Sig.ma are viable is an important and challenging matter. Sig.ma is built on *Sindice*, a web service that provides search and retrieval capabilities over semantic data. The web service uses *SIREn* at its core, an Information Retrieval search engine, to query an *information need* and to retrieve *relevant* documents.

In this report we presented data structures that are commonly used in Information Retrieval search engines. We also discussed about some of the focusing points for optimizing these structures in order to have more efficient and scalable IR search engines like SIREn. We presented AFOR, a compression method that provides both fast compression (increases index updates) and decompression (increases query throughput) speed, and yet with a high compression ratio. We proposed SkipBlock, a novel self-indexing model where some configurations carefully chosen provide faster random lookups from an inverted list and a more compact structure than the original Skip List model.

15.1 Summary of the Report

The flow of this report reflects the flow of a research work by (1) describing a problem and why current related work do not answer the requirements; (2) proposing a solution to the problem; and (3) proving the previous claims by performing comparative benchmarks. Following this pattern, we proposed two novel structures, where both have the common goal to reduce the amount of data read in order to increase the IO throughput.

Compression Technique compression techniques do not only aim at reducing the storage space, but also at increasing the IO access time. Reading/writing less data from/to disk with a high performance algorithm reduce the wasted time on IO access. Thus the performance of operations that directly depends on some data to process is improved. We proposed *AFOR*, a new compression class that can increase query throughput compared to other state of the art algorithms thanks to a more “close-to-data” compression.

Self-Indexing Technique query processing returns relevant information by applying some operations on the inverted lists. However not all the data that inverted lists have is necessary, thus reading or decoding such data is a wasted time. Self-indexing is a technique that allows to skip over portions of the inverted lists that are unnecessary for the processing of a query. We proposed a new self-indexing model, called *SkipBlock*, that aims to improve the original model Skip List by taking into consideration the compression algorithm used on the inverted lists. We will also present at *The* 33^{rd} *European Conference on Information Retrieval* (ECIR)[1] the paper which introduced the model (Appendix B).

15.2 Future Work

The Information Retrieval domain for Semantic Data covers not only what has been presented in this report but a wider area of problems. In the coming months, I will stay within DERI and work on different subjects. In this section I list a number of the possible future work.

- Finalize the AFOR implementation to make it into a production ready state.
- Continue the research on SkipBlock, which will consists in optimizing search strategies and finding more adapted ones.
- Implement a novel SIREn index structure that will improve query processing performance.
- Start researching on dynamic query processing.

15.3 Personal Benefits

My internship at DERI has been a source for new knowledge in many areas. I was able to deepen not only my skills in computer programming but also my scientific knowledge.

Computer Skills Thanks to this internship I was able to improve my skills on different programming languages: in JAVA since our project SIREn uses it, in script shell such as bash or Ruby when writing benchmarks automating scripts, in a text stream editor like *Sed*, very convenient for automate operations on large files such as logging or benchmarking results files.

Because SIREn is a system built to be highly efficient and scalable over millions of entities descriptions, any code that will be used within has to be well written. The engineer must take care of the memory and the CPU consumption so that the best performance is reached. Concerning JAVA there are many classes available to help the developer, for instance it is better to use the class `StringBuilder`[2] when operating of very large strings than to use the `String` type.

At last it is important to comment the code written, not only for people using it later on but also for

[1] ECIR: http://www.ecir2011.dcu.ie/

[2] `StringBuilder`: http://tinyurl.com/3xbkvw6

ourselves since it permits to know its structure or what are the possible optimizations. As part of commenting the code, writing meaningful descriptions in SVN logs helps to keep track of what was done and the reason of some changes.

Engineering Skills For the last months of my internship I was given a project (the SkipBlock model) to work on alone. This experience showed me the different points to take care of when managing projects, such as coordinating the development with some deadline. Also when implementing a solution, there are sometimes a difference between the model and the real results of the implementations. This leads to the necessity of taking decisions in order to understand why it is so and to be able to explain clearly the reasons. Moreover it is frequent when implementing under a deadline pressure that the code isn't optimized. In a short term this is not a problem, but in a long term it becomes one as a *technical debt*[3], since a messy code will end up in re-factoring.

Research Skills DERI made me aware of the challenges that we can expect from the research environment, such as a research dependent implementations which change a project flow, since time constraints cannot be put on tasks because the expecting difficulty and problems are still unknown. Moreover working on the SIREn project allowed me to take part into the publication process of scientific papers. These points as well as the scientific domain of DERI (Semantic Web) and of the project (IR plus highly efficient structures) gave me the desire to keep on working in the DI2 team for the following year.

Scientific Knowledge Thanks to this internship I was able to gain more knowledge on the interesting Information Retrieval domain. Moreover the internship made me aware of the Semantic Web infrastructure and of all the possibilities it provides.

Human Skills A project cannot be successful unless the communication between members is clear and efficient. A good communication flow allows members to know the big picture of the research, and what should be working on each individuals. DERI is a multi-cultural research institute, with people form all around the world. This working environment was a good basis for improving my English.

[3] Technical Debt: http://www.martinfowler.com/bliki/TechnicalDebt.html

Appendix A

Benchmarking Results

A.1 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 index structure with various compression algorithms. Tables A.1 have been used for generating the charts from Section 11.2. Tables A.2 have been used for generating the charts from Section 11.3.

	Time (s)		Sizes (GB)			
Method	Total	Opt	Doc	Frq	Pos	Total
AFOR-1	410	114	0.353	0.109	0.698	1.160
AFOR-2	409	128	0.340	0.093	0.655	1.088
FOR	443	128	0.426	0.178	0.811	1.415
PFOR	438	127	0.428	0.106	0.745	1.279
Rice	492	228	0.332	0.061	0.600	0.993
S9-64	462	124	0.356	0.103	0.662	1.121
VByte	433	132	0.402	0.291	0.714	1.407

(a) Wikipedia

	Time (s)		Sizes (GB)			
Method	Total	Opt	Doc	Frq	Pos	Total
AFOR-1	12337	4813	7.868	2.366	21.058	31.292
AFOR-2	13040	4571	7.387	2.016	19.492	28.895
FOR	12741	5888	9.115	3.882	26.780	39.777
PFOR	13972	5387	9.097	2.464	23.975	35.536
Rice	14099	7127	7.145	1.546	18.160	26.851
S9-64	13152	5414	7.892	2.197	19.711	29.800
VByte	12320	5092	8.630	5.610	22.063	36.303

(b) Blog

	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

(c) 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

(d) 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

(e) Sindice

Table A.1: Total indexing time, optimise time and index size.

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

(a) Value Query

	2 - AND			2 - OR			4 - AND			4 - OR			2 - Phrase			3 - Phrase		
Method	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB
DBpedia																		
AFOR-1	47.1	1.6	2.4	134.6	2.2	7.3	87.4	16.5	4.1	200.1	3.6	10.7	244.0	2.8	15.3	564.1	31.2	37.6
AFOR-2	64.0	2.1	2.2	132.5	2.7	6.8	103.0	15.3	3.8	220.0	10.3	9.9	282.3	17.0	14.2	594.4	15.2	34.7
AFOR-3	54.5	2.2	2.1	136.0	2.1	5.9	104.1	8.5	3.7	190.3	3.4	8.7	264.0	3.2	13.9	600.4	4.3	34.4
FOR	54.4	18.6	3.0	116.2	2.5	9.2	77.2	3.0	5.2	176.6	2.9	13.4	239.3	3.7	20.4	558.3	37.3	49.8
PFOR	61.3	4.7	3.1	146.1	2.4	8.7	117.1	4.2	5.3	199.3	3.9	12.7	249.3	3.5	18.0	578.2	32.6	43.3
Rice	107.0	2.4	2.3	312.2	3.2	6.8	192.8	3.2	3.9	475.5	12.2	9.8	677.0	5.2	13.2	1625.0	7.6	30.9
S-64	64.0	12.1	2.4	144.5	4.9	6.9	103.7	3.9	4.1	215.0	4.6	10.1	316.9	3.7	14.7	706.3	5.4	35.9
VByte	59.0	1.9	3.8	165.6	2.3	14.8	110.8	16.6	6.3	264.8	21.0	20.8	339.9	2.9	24.1	767.3	37.1	59.8
Geonames																		
AFOR-1	42.9	2.1	1.7	84.0	2.7	2.4	71.9	2.5	3.2	117.9	3.5	4.4	64.2	2.0	2.1	78.8	2.5	3.4
AFOR-2	55.6	18.7	1.7	91.2	1.9	2.3	85.9	19.1	3.1	129.8	3.6	4.3	59.8	2.7	2.0	90.1	3.3	3.2
AFOR-3	50.5	19.6	1.5	70.2	2.0	1.9	89.8	23.4	2.9	137.2	23.8	3.5	69.9	11.6	1.7	87.5	3.5	2.9
FOR	41.6	2.6	1.9	80.5	2.2	2.6	68.8	2.9	3.4	111.3	3.7	4.6	51.9	2.6	2.7	77.4	3.9	4.7
PFOR	56.3	3.2	2.0	81.0	2.9	2.7	94.1	2.8	3.5	137.5	4.6	4.7	67.4	3.1	2.8	98.2	3.5	4.5
Rice	97.5	2.7	1.9	158.1	5.7	2.5	165.2	4.2	3.4	272.8	2.8	4.6	120.8	2.6	2.2	173.5	3.2	3.6
S-64	60.6	21.1	2.1	83.7	3.0	2.6	96.0	3.8	3.8	168.9	4.0	4.9	75.4	17.7	2.2	99.3	3.4	3.5
VByte	57.9	17.6	3.9	91.9	2.5	6.8	95.2	3.4	6.8	159.2	3.5	12.5	82.7	2.2	4.5	116.1	24.5	6.9
Sindice																		
AFOR-1	55.5	1.9	2.1	192.9	2.4	8.2	77.8	2.4	3.9	311.1	3.1	14.4	310.3	4.0	19.0	1297.0	6.2	78.0
AFOR-2	53.3	1.6	1.9	229.2	3.0	7.5	105.1	11.3	3.5	330.7	32.3	13.2	341.0	4.0	17.4	1484.0	5.6	71.7
AFOR-3	52.1	1.9	1.8	180.2	2.5	5.5	88.7	2.5	3.3	291.2	3.2	10.0	334.8	2.9	16.6	1413.0	5.9	69.6
FOR	46.4	8.0	3.0	197.0	3.2	15.3	76.2	2.6	5.3	314.3	3.3	25.4	319.1	4.2	29.1	1304.0	7.1	115.9
PFOR	67.4	14.6	2.8	193.9	2.9	9.2	100.5	3.2	5.0	316.1	3.2	17.0	358.7	3.1	25.5	1348.0	7.4	102.3
Rice	100.4	2.3	2.4	481.3	3.8	10.9	170.5	3.4	4.1	797.8	30.6	18.5	825.6	5.3	19.2	3808.0	7.6	73.9
S-64	58.8	2.4	2.1	213.2	13.8	7.5	99.9	2.3	3.9	342.7	4.0	13.3	416.9	16.0	17.8	1724.0	7.8	73.7
VByte	58.8	12.8	3.8	311.3	3.0	25.9	97.8	2.2	6.5	478.1	53.0	42.5	438.4	4.1	32.7	1916.0	6.3	133.1

(b) Attribute Query

Table A.2: Query time execution using a node-based inverted index per query type, algorithm and dataset. We report for each query type the arithmetic mean (μ in millisecond), the standard deviation (σ in millisecond) and the total amount of data read during query processing (*MB* in megabyte).

	2 - AND			2 - OR			4 - AND			4 - OR			2 - Phrase			3 - Phrase		
Method	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB
Wikipedia																		
AFOR-1	146.3	3.1	5.7	244.1	7.5	5.8	203.3	18.0	11.3	553.9	11.6	11.4	971.6	10.3	62.6	2417.0	76.6	184.9
AFOR-2	153.0	11.9	5.4	262.0	6.0	5.4	212.0	3.9	10.6	558.8	3.5	10.7	970.3	5.0	58.9	2696.0	7.2	173.3
FOR	137.1	2.0	7.7	266.6	8.0	7.7	217.3	22.6	15.1	554.7	12.4	15.2	888.1	4.1	75.3	2429.0	17.1	224.2
PFOR	138.7	12.3	6.7	265.8	3.0	6.7	199.7	3.0	13.3	549.6	9.0	13.4	908.4	5.2	66.2	2518.0	6.1	195.2
Rice	258.1	8.0	5.0	372.5	2.6	5.0	439.9	10.4	9.8	788.0	9.2	9.9	2215.0	23.6	51.3	6234.0	9.4	149.5
S-64	152.6	6.4	5.7	277.7	7.2	5.7	229.0	15.9	11.2	573.6	10.5	11.3	1009.0	41.4	60.4	2790.0	47.8	177.1
VByte	164.7	2.0	8.6	286.8	5.6	8.7	258.6	14.4	16.8	597.3	12.0	17.0	1144.0	5.5	81.0	3113.0	77.5	240.6
Blog																		
AFOR-1	195.0	2.5	12.3	461.0	8.3	13.3	276.6	21.1	21.3	1034.0	5.6	25.4	3483.0	6.9	288.7	18934.0	309.1	1468.8
AFOR-2	212.8	13.2	11.4	518.5	5.9	12.3	298.7	13.2	19.8	1057.0	6.0	23.5	3805.0	65.5	265.5	20158.0	19.7	1334.8
FOR	199.4	9.7	15.4	502.7	8.7	16.7	290.8	29.2	26.6	1053.0	14.7	32.0	3606.0	7.1	362.4	18907.0	264.7	1904.4
PFOR	207.2	10.9	13.7	514.6	7.5	14.8	293.6	20.3	23.9	1033.0	5.5	28.6	3790.0	24.4	315.9	18144.0	345.2	1657.8
Rice	433.0	11.6	10.7	725.2	16.0	11.4	622.8	4.9	18.8	1471.0	12.7	22.1	9162.0	10.1	235.1	45689.0	30.5	1189.8
S-64	225.7	14.8	12.2	530.4	4.4	13.1	313.2	10.1	21.2	1100.0	5.5	25.1	4005.0	9.7	273.0	21260.0	314.4	1370.6
VByte	248.7	19.4	17.1	536.4	22.1	18.6	376.4	33.4	29.1	1200.0	11.8	35.0	4400.0	7.3	355.3	21615.0	25.9	1762.6

Table A.3: Query time execution using a traditional inverted index per query type, algorithm and dataset. We report for each query type the arithmetic mean (μ in millisecond), the standard deviation (σ in millisecond) and the total amount of data read during query processing (*MB* in megabyte).

A.2 Results of the Scalability Benchmark

This appendix provides the table containing the results of the scalability benchmark. Table A.4 has been used for generating the charts from Section 12.3.

	1			2			4			8			16		
Selectivity	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB	μ	σ	MB
Small															
LMH	66.9	0.2	165.7	198.9	6.8	55.8	292.3	3.6	31.6	145.5	6.4	50.2	94.1	4.4	65.7
MH	24.6	0.8	424.8	112.9	4.1	90.3	97.3	0.6	121.6	90.7	3.4	110.7	85.7	3.6	91.3
Medium															
LMH	56.2	0.2	188.0	209.2	7.8	52.5	272.6	6.2	33.2	191.8	10.7	18.7	120.5	0.7	40.0
MH	17.5	0.1	301.8	116.0	3.4	80.8	158.0	6.5	72.6	109.0	4.8	74.1	103.8	5.7	70.8
Large															
LMH	28.1	0.1	377.4	244.5	1.1	51.2	230.8	1.2	41.6	152.3	8.2	50.2	105.4	4.0	58.4
MH	20.3	0.1	543.3	128.7	0.5	100.1	106.4	0.7	103.7	108.0	2.3	95.2	58.2	0.7	96.7

Table A.4: Query rate per dataset, term selectivity and query size. We report for each query size the arithmetic mean (μ 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).

A.3 Results of the Self-Indexing Benchmarks

This appendix provides the table containing the results of the self-indexing benchmarks. The Table A.5 and Table A.6 have been used for generating the charts from Section 13.2.2. In both tables, the columns report the following information:

- *Operands* indicates from which frequency groups the inverted lists were taken from.
- $|I|$ stands for the interval length.
- *MB* reports the number in MBytes of data read from disk when searching in the self-indexing structure.
- *Scans* reports the number of records scanned.
- *Size* reports the structure's size.
- *Time* indicates the average runtime of the operation
- *Ops* gives the number of search operations.

Operands	$\lvert I\rvert$	MB	Scans	Size (in MB)	Time	Ops
HIGH:HIGH	16	12.02	50 197 056	150.15	3.400 s ± 7.932 ms	55 766 922
	32	7.36	52 524 966	68.43	2.492 s ± 12.154 ms	55 882 733
	256	2.97	64 766 407	15.0	2.484 s ± 10.052 ms	65 538 236
	512	1.66	72 983 836	7.47	2.541 s ± 77.962 ms	73 415 418
	1024	0.95	85 129 635	3.73	2.619 s ± 5.970 ms	85 376 552
	4096	0.44	120 713 279	0.941	3.025 s ± 6.837 ms	120 826 073
HIGH:LOW	16	0.004	447	84.49	363.501 us ± 101.008 us	1 398
	32	0.006	670	38.2	326.611 us ± 100.244 us	2 088
	256	0.024	4 315	8.45	545.605 us ± 115.329 us	9 659
	512	0.022	8 407	4.2	579.102 us ± 606.765 us	13 397
	1024	0.032	16 207	2.1	735.205 us ± 129.810 us	24 296
	4096	0.123	55 364	0.526	2.433 s ± 288.585 ms	87 049

(a) Skip List self-indexing model.

Operands	$\lvert I\rvert$	C	I_2					I_3				
			MB	Scans	Size (in MB)	Time	Ops	MB	Scans	Size (in MB)	Time	Ops
HIGH:HIGH	16	B=4 p=4	14.10	47 270 398	247.17	3.631 s ± 13.698 ms	53 390 244	40.83	33 665 878	959.1	3.746 s ± 11.002 ms	53 680 730
	32	B=8 p=4	9.16	51 058 497	136.13	2.650 s ± 9.693 ms	54 471 140	23.87	42 187 630	491.09	3.795 s ± 17.746 ms	53 203 556
	256	B=32 p=8	2.25	64 576 706	18.83	2.844 s ± 36.334 ms	65 153 966	7.60	51 063 586	93.35	2.646 s ± 7.886 ms	54 160 474
	512	B=64 p=8	1.30	72 893 207	9.39	2.517 s ± 7.389 ms	73 224 689	4.99	54 730 109	49.9	2.627 s ± 7.356 ms	56 475 877
	1024	B=64 p=16	0.74	85 080 396	4.28	2.925 s ± 8.417 ms	85 264 977	4.81	54 730 557	41.53	2.582 s ± 8.124 ms	56 514 322
	4096	B=256 p=16	0.22	120 676 635	1.08	3.581 s ±84.845 ms	120 729 476	2.30	64 576 744	16.69	3.509 s ± 59.808 ms	65 173 924
HIGH:LOW	16	B=4 p=4	0.0031	329	137.99	581.592 us ± 247.832 us	932	0.0034	150	538.87	640.820 us ± 133.024 us	862
	32	B=8 p=4	0.0026	489	76.67	521.191 us ± 199.168 us	990	0.0028	234	276.38	585.107 us ± 112.912 us	820
	256	B=32 p=8	0.0023	2 733	10.65	352.783 us ± 95.043 us	3 176	0.0027	521	52.38	381.714 us ± 102.162 us	1 102
	512	B=64 p=8	0.0023	5 292	5.3	432.813 us ± 91.323 us	5 718	0.0027	982	26.84	370.581 us ± 244.365 us	1 539
	1024	B=64 p=16	0.0027	8 100	2.41	453.247 us ± 117.707 us	8 605	0.0034	982	22.05	311.572 us ± 95.760 us	1 739
	4096	B=256 p=16	0.0025	28 826	0.60	668.652 us ± 600.146 us	3 055	0.0034	2 988	9.425	343.726 us ± 97.064 us	3 668

(b) SkipBlock self-indexing model. C gives the structure's configuration with the size of a block B and the inverse probability p.

Table A.5: Conjunction operation results.

Operands	$\|I\|$	MB	Scans	Size (in MB)	Time	Ops
HIGH:HIGH	16	6,168	36 798 956	150.15	17.078 s ± 147.674 ms	39 711 326
	32	3,615	37 906 536	68.43	18.508 s ± 23.425 ms	39 451 909
	256	1,248	42 539 161	15.0	17.122 s ± 25.578 ms	42 863 287
	512	0,765	44 390 563	7.47	17.128 s ± 73.775 ms	44 588 824
	1024	0,466	46 776 308	3.73	17.297 s ± 93.803 ms	46 896 776
	4096	0,194	54 212 443	0.941	16.836 s ± 45.173 ms	54 261 457
LOW:HIGH	16	0,002	57	68.37	201.953 us ± 72.146 us	429
	32	0,003	121	31.63	167.969 us ± 78.125 us	662
	256	0,012	1 017	6.82	276.294 us ± 114.937 us	3 348
	512	0,018	1 529	3.4	345.850 us ± 464.155 us	5 574
	1024	0,022	3 577	1.7	444.019 us ± 114.084 us	9 140
	4096	0,078	11 769	0.432	1.320 ms ± 163.548 us	31 796

(a) Skip List self-indexing model.

Operands	$\|I\|$	C	I_2					I_3				
			MB	Scans	Size (in MB)	Time	Ops	MB	Scans	Size (in MB)	Time	Ops
HIGH:HIGH	16	B=4 p=4	7,896	35 261 542	247.17	20.375 s ± 201.675 ms	38 711 122	25,704	29 387 051	959.1	20.421 s ± 107.201 ms	42 111 448
	32	B=8 p=4	4,919	37 132 956	136.13	20.849 s ± 103.259 ms	38 938 380	14,193	32 967 483	491.09	21.074 s ± 43.166 ms	39 573 145
	256	B=32 p=8	0,936	42 438 570	18.83	20.498 s ± 77.985 ms	42 677 604	3,996	37 138 012	93.35	20.083 s ± 97.600 ms	38 722 432
	512	B=64 p=8	0,517	44 344 695	9.39	19.838 s ± 100.887 ms	44 475 347	2,513	38 961 314	49.9	19.741 s ± 49.508 ms	39 795 225
	1024	B=64 p=16	0,296	46 759 073	4.28	20.694 s ± 56.040 ms	46 831 641	2,376	38 961 698	41.53	20.269 s ± 39.224 ms	39 771 744
	4096	B=256 p=16	0,096	54 205 740	1.076	15.826 s ± 228.631 ms	54 228 258	0,919	42 438 570	16.69	15.783 s ± 84.509 ms	42 675 368
LOW:HIGH	16	B=4 p=4	0,001 49	58	113.66	329.077 us ± 111.135 us	316	0,001 55	18	437.66	366.797 us ± 176.942 us	294
	32	B=8 p=4	0,001 29	90	62.01	336.328 us ± 129.074 us	314	0,001 34	34	223.58	354.395 us ± 105.327 us	274
	256	B=32 p=8	0,001 26	858	8.52	231.824 us ± 163.947 us	1 070	0,001 35	90	42.8	227.954 us ± 93.254 us	337
	512	B=64 p=8	0,001 33	1 114	4.27	227.222 us ± 409.992 us	1 337	0,001 41	218	24.15	217.444 us ± 65.653 us	467
	1024	B=64 p=16	0,001 72	2 138	1.95	198.083 us ± 77.898 us	2 428	0,001 86	218	20.43	186.499 us ± 72.585 us	566
	4096	B=256 p=16	0,001 48	8 282	0.494	273.047 us ± 370.889 us	8 538	0,001 69	858	7.57	185.840 us ± 75.130 us	1 163

(b) SkipBlock self-indexing model. C gives the structure's configuration with the size of a block B and the inverse probability p.

Table A.6: Exclusion operation results.

Appendix B

SkipBlock: Self-Indexing for Block-Based Inverted List

This appendix provides the current version of the short paper that shall be presented at the ECIR[1] conference, which is the basis of the SkipBlock implementation presented in the Section **??**.

[1] ECIR: http://www.ecir2011.dcu.ie/

SkipBlock: Self-Indexing for Block-Based Inverted List

Stéphane Campinas, Renaud Delbru and Giovanni Tummarello

Digital Enterprise Research Institute,
National University of Ireland, Galway
Galway, Ireland

Abstract. In large web search engines the performance of Information Retrieval systems is a key issue. Block-based compression methods are often used to improve the search performance, but current self-indexing techniques are not adapted to such data structure and provide sub-optimal performance. In this paper, we present SkipBlock, a self-indexing model for block-based inverted lists. Based on a cost model, we show that it is possible to achieve significant improvements on both search performance and structure's space storage.

1 Introduction

The performance of Information Retrieval systems is a key issue in large web search engines. The use of compression techniques and self-indexing inverted files [8] is partially accountable for the current performance achievement of web search engines. On the one hand, compression maximises IO throughput [3] and therefore increases query throughput. On the other hand, self-indexing inverted files [8] enables the intersection of inverted lists in sub-linear time.

Nowadays efficient inverted index compression methods tend to have a block-based approach [6,10,1]. An inverted list is divided into multiple non-overlapping blocks of records. The coding is then done a block at a time and independently from the others. While block-based coding approaches provide incontestable benefits, the self-indexing method [8] achieves only sub-optimal performance on block-based inverted lists. The reason is that the self-indexing technique disregards the block-based structure of the list which however can be used for designing a more efficient self-indexing as we will show in this paper.

In this paper we present an approach for self-indexing of block-based inverted lists. We demonstrate the benefits of our block-based self-indexing technique by comparing it against the original self-indexing approach based on a cost model. In Section 2 we first review the original self-indexing technique based on the Skip Lists data structure, before presenting in Section 3 our approach. Section 4 discuss the problem of searching within Skip Lists intervals. In Section 5 we define a cost model and compare four implementations of our model against the original model. In Section 6 we recall the main finding of the research and the task that remains.

1.1 Related Work

The Skip Lists data structure is introduced by [9] as a probabilistic alternative to balanced trees and is shown in [5] to be as elegant and easier to use than binary search trees. Such a structure is later employed for self-indexing of inverted lists in [8]. Self-indexing inverted list enables a sub-linear complexity in average when intersecting two inverted lists. [2] proposes a way to compress efficiently a Skip Lists directly into an inverted list and shows that it is possible to achieve a substantial performance improvement. In [4], the authors introduce a method to place skips optimally given the knowledge of the query distribution. [7] presents a generalized Skip Lists data structure for concurrent operations. In this paper, we introduce a new model for self-indexing of block-based inverted lists which is based on an adaptation of the Skip Lists data structure. Our work is orthogonal to the previous works, since they might be extended to our model.

2 Background: Self-Indexing for Inverted Lists

An inverted list is an ordered list of compressed records (e.g., documents identifiers). When intersecting two or more inverted lists, we often need to access random records in those lists. The basic approach is to scan linearly the lists to find them. Such an operation is not optimal and can be reduced to sub-linear complexity in average by the use of the self-indexing approach [8]. Self-indexing is based on Skip Lists to build a sparse index over the inverted lists and to provide fast record lookups. In this section, we first present the Skip Lists model and its associated search algorithm. We finally discuss the effect of the probabilistic parameter with respect to the Skip Lists data structure and search complexity.

2.1 The Skip Lists Model

Skip Lists are used to index records in an inverted list at regular interval. These indexing points, called *synchronization points*, are organized into a hierarchy of linked lists, where a linked list at level $i+1$ has a probability p to index a record of the linked list at level i. The probabilistic parameter p is fixed in advance and indicates the *interval* between each synchronization point at each level. For example in Figure 1, a synchronization point is created every $\frac{1}{p^1} = 16$ records at level 1, every $\frac{1}{p^2} = 256$ records at level 2, and so on. In addition to the pointer to the next synchronization point on a same level, a synchronization point at level $i+1$ has a pointer to the same synchronization point at level i. For example in Figure 1, the first synchronization point at level 3 (i.e., for the record 4096) has a pointer to the level 2 which has itself a pointer to the level 1. This hierarchical structure enables to quickly find a given record using a top-down search strategy.

Given the probabilistic parameter p and the size n of an inverted list, we can deduce two characteristics of the resulting Skip Lists data structure: (1) the expected number of levels and (2) the size, i.e., the total number of synchronization points. The number of levels in the Skip Lists is defined by $L(n) = \lfloor \ln_{\frac{1}{p}}(n) \rfloor$, which is the maximum as stated in [9]. The total number of synchronization points is given by $S(n) = \sum_{i=1}^{L(n)} \lfloor n \times p^i \rfloor$, which sums up the number of synchronization points expected at each level.

2.2 Skip Lists Search Algorithm

Searching a Skip Lists enables to retrieve an interval containing the target record and is performed using a top-down strategy. The search within an interval is discussed in Section 4. The search walk starts at the head of the top list, and performs a linear walk on a level as long as the target is greater than a synchronization point. The walk goes down one level if and only if the target is lower than the current synchronisation point. The search finishes when the current synchronization point is (a) equal to the target, or (b) on the bottom level and greater than the target. The reached interval then contains the target. The search complexity is defined by the number of steps necessary to find the interval holding the target. In the worst case, the number of steps at each level is at most $\frac{1}{p}$ in at most $L(n)$ levels. Consequently, the search complexity is $\frac{L(n)}{p}$.

Figure 1 depicts with a solid line the search path in a Skip Lists with $p = \frac{1}{16}$ and $L(n) = 3$ levels to the record 8195. At the top of the Skip Lists, we walk to the record 8192. Then we go down to level 1 and stop because the current synchronization point (i.e., the record 8208) is greater than the target. At this point, we know that the target record is in the next interval on the inverted list.

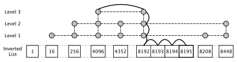


Fig. 1: Skip Lists with $p = \frac{1}{16}$. Dashed lines denote pointers between synchronization points. The solid line shows the search path to the record 8195.

2.3 Impact of the Probabilistic Parameter

In this section, we discuss the consequences of the probabilistic parameter on the Skip Lists data structure. Table 1a reports for low (i.e., $\frac{1}{1024}$) and high (i.e., $\frac{1}{2}$) probabilities (1) the complexity $\frac{L(n)}{p}$ to find the interval containing the target record, and (2) the size $S(n)$ of the Skip Lists structure. There is a trade-off to achieve when selecting p: a high probability provides a low search complexity but at a larger space cost, and a low probability reduces considerably the required space at the cost of higher search complexity. The SkipBlock model presents a way to reduce even more the search complexity in exchange of a larger data structure.

$\lvert I\rvert$	2	16	64	128	1024
$S(n)$	99 999 988	6 666 664	1 587 300	787 400	97 751
C	54	112	320	512	3072

(a) Skip Lists with $|I| = \frac{1}{p}$.

$\lvert I\rvert$	16		64		128		1024	
p:$\lvert B\rvert$	$\frac{1}{4}$:4	$\frac{1}{8}$:2	$\frac{1}{4}$:16	$\frac{1}{8}$:8	$\frac{1}{4}$:32	$\frac{1}{8}$:16	$\frac{1}{4}$:256	$\frac{1}{8}$:128
$S_B(n)$	8 333 328	7 142 853	2 083 328	1 785 710	1 041 660	892 853	130 203	111 603
C	48	64	44	56	40	56	36	48

(b) SkipBlock with $|I| = \frac{|B|}{p}$.

Table 1: Search and size costs of Skip Lists and SkipBlock with $n = 10^8$. $|I|$ stands for an interval length. C reports the search complexity to find an interval (Sections 2.2 and 3.2).

3 SkipBlock: A Block-Based Skip Lists Model

In this section, we introduce the SkipBlock model and present its associated search algorithm. Finally we discuss how the SkipBlock model offers finer control over the Skip Lists data structure in order to trade search cost against storage cost.

3.1 The SkipBlock Model

The SkipBlock model operates on *blocks* of records of a fixed size, in place of the records themselves. Consequently, the probabilistic parameter p is defined with respect to a block unit. A synchronization point is created every $\frac{1}{p^i}$ blocks on a level i, thus every $\frac{|B|}{p^i}$ records where $|B|$ denotes the block size. A synchronization point links to the first record of a block interval. Compared to Figure 1, a SkipBlock structure with $p = \frac{1}{8}$ and $|B| = 2$ also has an interval of $\frac{|B|}{p^1} = 16$ records. However, on level 2, the synchronization points are separated by $\frac{|B|}{p^2} = 128$ instead of 256 records. We note that with $|B| = 1$, the SkipBlock model is equivalent to the original Skip Lists model, and therefore it is a generalization. The number of levels is defined by $L_B(n) = \left\lfloor \ln_{\frac{1}{p}} \left(\frac{n}{|B|} \right) \right\rfloor$ and the size by $S_B(n) = \sum_{i=1}^{L_B(n)} \left\lfloor \frac{n \times p^i}{|B|} \right\rfloor$.

3.2 SkipBlock Search Algorithm

With the block-based Skip Lists model, the search algorithm returns an interval of blocks containing the target record. We discuss the search in that interval in Section 4. The search walk is identical to the one presented in Section 2.2: we walk from the top to the bottom level, and compare at each step the current synchronization point with the target. The walk stops at the same termination criteria as in Skip Lists. The search complexity in the worst case becomes $\frac{L_B(n)}{p}$.

3.3 Impact of the Probability and of the Block's Size

The SkipBlock model provides two parameters to control its Skip Lists structure: the probabilistic parameter p and the block size $|B|$. The block size parameter enables more control over the Skip Lists structure. For example, to build a structure with an interval of length 64, the original Skip Lists model proposes only one configuration given by $p = \frac{1}{64}$. For this same interval length, SkipBlock proposes all the configurations that verify the equation $\frac{|B|}{p} = 64$. Table 1b reports statistics of some SkipBlock configurations for the same interval lengths as in Table 1a. Compared to Skip Lists on a same interval length, SkipBlock shows a lower search complexity in exchange of a larger structure.

4 Searching Records in an Interval

The Skip Lists and SkipBlock techniques enables the retrieval of an interval given a target record. The next step is to find the target record within that interval. A first strategy (S1) is to linearly scan all the records within that interval until the target is found. Its complexity is therefore $O(|I|)$.

SkipBlock takes advantage of the block-based structure of the interval to perform more efficient search strategy. We define here four additional strategies for searching a block-based interval, parameterized to a probability p. The second strategy (S2) performs (a) a linear scan over the blocks of the interval to find the block holding the target and (b) a linear scan of the records of that block to find the target. The search complexity is $\frac{1}{p} + |B|$ with $\frac{1}{p}$ denoting the linear scan over the blocks and $|B|$ the linear scan over the records of one block. Similarly to S2, the third strategy (S3) performs the step (a). Then, it uses a inner-block Skip Lists structure, restricted to one level only, to find the target. The complexity is $\frac{1}{p} + \frac{1}{q} + \lfloor |B| \times q \rfloor$ with q the probability of the inner Skip Lists. In contrast to S3, the fourth strategy (S4) uses a non-restricted inner-block Skip Lists structure. The complexity is $\frac{1}{p} + \frac{L(|B|)+1}{q}$ with q the inner Skip Lists probability. The fifth one (S5) builds a Skip Lists structure on the whole interval instead of on a block. Its complexity is then $\frac{L\left(\frac{|B|}{p}\right)+1}{q}$, with q the inner Skip Lists probability. The strategies S3, S4 and S5 are equivalent to S2 when the block size is too small for creating synchronization points.

5 Cost-Based Comparison

In this section, we define a cost model that we use to compare five SkipBlock implementations and the original Skip Lists implementation.

Cost Model For both the Skip Lists and the SkipBlock, we define a cost model by (a) the cost to search for the target, and (b) the cost of the data structure's size. The search cost consists in the number of synchronization points traversed to reach the interval containing the target, plus the number of records scanned in that interval to find the target. The size cost consists in the total number of synchronization points in the data structure, including the additional ones for S3, S4 and S5 in the intervals.

Implementations We define as the baseline implementation, denoted I_1, the Skip Lists model using the strategy (*S1*). We define five implementations of the SkipBlock model, denoted by I_2, I_3, I_4, I_5 and I_6, based on the five interval search strategies, i.e., *S1*, *S2*, *S3*,*S4* and *S5* respectively. The inner Skip Lists in implementations I_4, I_5 and I_6 is configured with probability $q = \frac{1}{16}$. The inner Skip Lists in I_5 and I_6 have at least 2 levels. The size costs are $S(n)$ for I_1, $S_B(n)$ for I_2, $S_B(n) + \frac{n}{|B|}$ for I_3, $S_B(n) + \lfloor n \times q \rfloor$ for I_4, $S_B(n) + \frac{S(|B|) \times n}{|B|}$ for I_5 and $S_B(n) + \frac{S(p \times |B|) \times n}{p \times |B|}$ for I_6.

Comparison With respect to the SkipBlock model, we tested all the possible configurations for a given interval length. We report that all of them were providing better search cost than the baseline. We only report in Table 2 the configurations providing the best search cost given an interval length with the associated size cost. We observe that I_2 already provides better search cost than the baseline I_1 using the same search strategy *S1*, in exchange of a larger size cost. The other implementations, i.e., I_3, I_4, I_5 and I_6 which use a more efficient interval search strategies further decrease the search cost. In addition, their size cost decreases significantly with the size of the interval. On a large interval (512), I_4 is able to provide a low search cost (64) while sustaining a small size cost (6.5e6). The inner Skip Lists of I_5 and I_6 are built on lists too small to provide benefits. To conclude, I_4 seems to provide a good compromise between search cost and size cost with large intervals.

$\|I\|$	8						16						512						1152					
	I_1	I_2	I_3	I_4	I_5	I_6	I_1	I_2	I_3	I_4	I_5	I_6	I_1	I_2	I_3	I_4	I_5	I_6	I_1	I_2	I_3	I_4	I_5	I_6
SC	72	56	54				112	62	56				1536	548	120	64	86	84	3456	1186	154	74	84	81
ZC×e6	14.3	16.7	50.0				6.7	12.5	25.0				0.2	0.3	1.8	6.5	7.0	6.9	0.09	0.17	1.7	6.4	6.6	6.7

Table 2: Search (i.e., SC) and size (i.e., ZC, in million) costs with $n = 10^8$. SkipBlock implementations report the best search cost with the associated size cost.

6 Conclusion and future work

We presented SkipBlock, a self-indexing model for block-based inverted lists. The SkipBlock model extends the original Skip Lists model and provides a backbone for developing efficient interval search strategies. Compared to the original Skip Lists model, SkipBlock can achieve a lower search complexity and a smaller data structure size. In addition, SkipBlock allows finer control over the Skip Lists data structure and so additional possibilities for trading search costs against storage costs. Future work will focus on real world data benchmarks in order to assess the performance benefits of the SkipBlock model.

References

1. Anh, V.N., Moffat, A.: Index compression using 64-bit words. Softw. Pract. Exper. 40(2), 131–147 (2010)
2. Boldi, P., Vigna, S.: Compressed perfect embedded skip lists for quick inverted-index lookups. In: In Proc. SPIRE 2005, Lecture Notes in Computer Science. pp. 25–28. SpringerVerlag (2005)
3. Büttcher, S., Clarke, C.L.A.: Index compression is good, especially for random access. In: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management - CIKM '07. pp. 761–770. ACM Press, New York, New York, USA (2007)
4. Chierichetti, F., Lattanzi, S., Mari, F., Panconesi, A.: On placing skips optimally in expectation. In: WSDM '08: Proceedings of the international conference on Web search and web data mining. pp. 15–24. ACM, New York, NY, USA (2008)
5. Dean, B.C., Jones, Z.H.: Exploring the duality between skip lists and binary search trees. In: ACM-SE 45: Proceedings of the 45th annual southeast regional conference. pp. 395–399. ACM, New York, NY, USA (2007)
6. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: In proceedings of IEEE International Conference on Data Engineering. pp. 370–379 (1998)
7. Messeguer, X.: Skip trees, an alternative data structure to skip lists in a concurrent approach. ITA pp. 251–269 (1997)
8. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst. 14(4), 349–379 (1996)
9. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
10. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: ICDE '06: Proceedings of the 22nd International Conference on Data Engineering. p. 59. IEEE Computer Society, Washington, DC, USA (2006)

Bibliography

[1] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of the 6th International Conference on Database Theory*, pages 1–18, 1997.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, 1996.

[3] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: a system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering*, pages 5–16, San Jose, California, 2002. IEEE Comput. Soc.

[4] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, 2005.

[5] Vo Ngoc Anh and Alistair Moffat. Structured Index Organizations for High-Throughput Text Querying. In *Proceedings of the 13th International Conference of String Processing and Information Retrieval*, pages 304–315. Springer, 2006.

[6] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[7] Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges on Distributed Web Retrieval. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 6–20. IEEE, 2007.

[8] Dave Beckett and Jan Grant. Semantic Web Scalability and Storage: Mapping Semantic Web Data with RDBMSes. SWAD-Europe deliverable, W3C, January 2003.

[9] Kevin Beyer, Stratis D. Viglas, Igor Tatarinov, Jayavel Shanmugasundaram, Eugene Shekita, and Chun 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*, pages 204–215, New York, NY, USA, 2002. ACM.

[10] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering*, pages 431–440. IEEE Comput. Soc, 2002.

[11] Paolo Boldi and Sebastiano Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *In Proc. SPIRE 2005, Lecture Notes in Computer Science*, pages 25–28. Springer–Verlag, 2005.

[12] Brent Boyer. Robust Java benchmarking, 2008.

[13] Kevin Burton, Akshay Java, and Ian Soboroff. The ICWSM 2009 Spinn3r Dataset. In *Third Annual Conference on Weblogs and Social Media (ICWSM 2009)*, San Jose, CA, May 2009. AAAI. http://icwsm.org/2009/data/.

[14] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[15] Gong Cheng, Weiyi Ge, and Yuzhong Qu. Falcons: searching and browsing entities on the semantic web. In *Proceeding of the 17th international conference on World Wide Web*, pages 1101–1102. ACM, 2008.

[16] Flavio Chierichetti, Silvio Lattanzi, Federico Mari, and Alessandro Panconesi. On placing skips optimally in expectation. In *WSDM '08: Proceedings of the international conference on Web search and web data mining*, pages 15–24, New York, NY, USA, 2008. ACM.

[17] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. XSEarch: a semantic search engine for XML. In *Proceedings of the 29th international conference on Very large data bases - VLDB '2003*, pages 45–56. VLDB Endowment, 2003.

[18] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, pages 395–399, New York, NY, USA, 2007. ACM.

[19] Li Ding, Rong Pan, Tim Finin, Anupam Joshi, Yun Peng, and Pranam Kolari. Finding and Ranking Knowledge on the Semantic Web. In *Proceedings of the 4th International Semantic Web Conference*, pages 156–170, 2005.

[20] Xin Dong and Alon Halevy. Indexing dataspaces. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07*, page 43, 2007.

[21] Vuk Ercegovac, David J. DeWitt, and Raghu 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*, pages 313–324. VLDB Endowment, 2005.

[22] George H. L. Fletcher, Jan Van Den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Towards a theory of search queries. In *Proceedings of the 12th International Conference on Database Theory*, pages 201—-211, 2009.

[23] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering*, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

[24] R. Guha, Rob McCool, and Eric Miller. Semantic search. In *Proceedings of the 12th international conference on World Wide Web*, pages 700–709, 2003.

[25] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*, pages 16–27, New York, New York, USA, 2003. ACM Press.

[26] Andreas Harth, Aidan Hogan, Jürgen Umbrich, and Stefan Decker. SWSE: Objects before documents! In *Proceedings of the Billion Triple Semantic Web Challenge, 7th International Semantic Web Conference*, 2008.

[27] Vagelis Hristidis and Yannis Papakonstantinou. Discover: keyword search in relational databases. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 670–681, 2002.

[28] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases*, pages 505–516, Trondheim, Norway, 2005.

[29] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. NAGA: harvesting, searching and ranking knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, pages 1285–1288, Vancouver, Canada, 2008.

[30] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, pages 903–914, Vancouver, Canada, 2008.

[31] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[32] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, page 563, Chicago, IL, USA, 2006. ACM Press.

[33] Ziyang Liu, Jeffrey Walker, and Yi Chen. XSeek: a semantic XML search engine using keywords. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1330–1333, 2007.

[34] Federica Mandreoli, Riccardo Martoglia, Giorgio Villani, and Wilma Penzo. Flexible query answering on graph-modeled data. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 216—-227, Saint Petersburg, Russia, 2009. ACM.

[35] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *ITA*, pages 251–269, 1997.

[36] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[37] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[38] Manfred Hauswirth Stefan Decker. Enabling Networked Knowledge. Technical report, Digital Enterprise Research Institute, 2008.

[39] Giovanni Tummarello, Richard Cyganiak, Michele Catasta, Scyzmon Danielczyk, Renaud Delbru, and Stefan Decker. Sig.ma: Live views on the Web of Data. *Journal of Web Semantics*, 2010.

[40] Rossano Venturini and Fabrizio 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.

[41] Haofen Wang, Qiaoling Liu, Thomas Penin, Linyun Fu, Lei Zhang, Thanh Tran, Yong Yu, and Yue 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):177–188, 2009.

[42] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, pages 147–154, New York, NY, USA, 2009. ACM.

[43] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World Wide Web - WWW '09*, pages 401–410, New York, New York, USA, 2009. ACM Press.

[44] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*, pages 387–396, New York, New York, USA, 2008. ACM Press.

[45] M. Zukowski, S. Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59, Washington, DC, USA, 2006. IEEE Computer Society.

List of Listings

List of Figures

List of Tables