Big Data to Enable Global Disruption of the Grapevine-powered Industries

D3.3 - Distributed Indexing Components

DELIVERABLE NUMBER	D2.3
DELIVERABLE TITLE	Distributed Indexing Components
RESPONSIBLE AUTHOR	Rossano Venturini (CNR)

Co-funded by the Horizon 2020
Framework Programme of the European Union

GRANT AGREEMENT N.	780751
PROJECT ACRONYM	BigDataGrapes
PROJECT FULL NAME	Big Data to Enable Global Disruption of the Grapevine-powered industries
STARTING DATE (DUR.)	01/01/2018 (36 months)
ENDING DATE	31/12/2020
PROJECT WEBSITE	http://www.bigdatagrapes.eu/
COORDINATOR	Pythagoras Karampiperis
ADDRESS	110 Pentelis Str., Marousi, GR15126, Greece
REPLY TO	pythk@agroknow.com
PHONE	+30 210 6897 905
EU PROJECT OFFICER	Mr. Riku Leppanen
WORKPACKAGE N. | TITLE	WP3 | Data & Semantics Layer
WORKPACKAGE LEADER	ONTOTEXT
DELIVERABLE N. | TITLE	D3.3 | Distributed Indexing Components
RESPONSIBLE AUTHOR	Rossano Venturini (CNR)
REPLY TO	Rossano.Venturini@unipi.it
DOCUMENT URL	http://www.bigdatagrapes.eu/
DATE OF DELIVERY (CONTRACTUAL)	30 September 2018 (M9)
DATE OF DELIVERY (SUBMITTED)	28 September 2018 (M9)
VERSION | STATUS	1.0 | Final
NATURE	Demonstrator (DEM)
DISSEMINATION LEVEL	Public (PU)
AUTHORS (PARTNER)	Rossano Venturini (CNR), Raffaele Perego (CNR), Milena Yankova (ONTOTEXT), Pythagoras Karampiperis (Agroknow)
CONTRIBUTORS	Vladimir Alexiev (ONTOTEXT), Panagiotis Zervas (Agroknow), Sabine Karen Yemadje Lammoglia (INRA), Arnaud Charleroy (INRA), Pascal Neveu (INRA), Aikaterini Kasimati (AUA), Maritina Stavrakaki (AUA)
REVIEWER	Stefan Scherer (GEOCLEDIAN)

VERSION	MODIFICATION(S)	DATE	AUTHOR(S)
0.1	Table of Contents	07/09/2018	Rossano Venturini (CNR)
0.5	Initial version	12/09/2018	Rossano Venturini (CNR)
0.8	Input from partners	14/9/2018	Vladimir Alexiev (ONTOTEXT), Panagiotis Zervas (Agroknow), Sabine Karen Yemadje Lammoglia (INRA), Arnaud Charleroy (INRA), Pascal Neveu (INRA), Aikaterini Kasimati (AUA), Maritina Stavrakaki (AUA)
0.9	Internal Review	21/09/2018	Stefan Scherer (GEOCLEDIAN)
1.0	Final edits after internal review	25/09/2018	Rossano Venturini (CNR), Raffaele Perego (CNR), Milena Yankova (ONTOTEXT), Pythagoras Karampiperis (Agroknow)

PARTICIPANTS	CONTACT
Agroknow IKE (Agroknow, Greece)	Pythagoras Karampiperis Email: pythk@agroknow.com
Ontotext AD (ONTOTEXT, Bulgaria)	Todor Primov Email: todor.primov@ontotext.com
Consiglio Nazionale Delle Richerche (CNR, Italy)	Raffaele Perego Email: raffaele.perego@isti.cnr.it
Katholieke Universiteit Leuven (KULeuven, Belgium)	Katrien Verbert Email: katrien.verbert@cs.kuleuven.be
Geocledian GmbH (GEOCLEDIAN Germany)	Stefan Scherer Email: stefan.scherer@geocledian.com
Institut National de la Recherché Agronomique (INRA, France)	Pascal Neveu Email: pascal.neveu@inra.fr
Agricultural University of Athens (AUA, Greece)	Katerina Biniari Email: kbiniari@aua.gr
Abaco SpA (ABACO, Italy)	Simone Parisi Email: s.parisi@abacogroup.eu
APIGAIA (APIGEA, Greece)	Eleni Foufa Email: Foufa-e@apigea.com

ACRONYMS LIST

BDG	Big Data Grapes
BIC	Binary Interpolative Coding
D-GAPS	Delta Gaps
GDBMS	Graph Data Base Management System
LSM-tree	Log-Structured Merge-tree
OWL	Web Ontology Language
PEF	Partitioned Elisa Fano
RDF	Resource Description Framework
RDFS	RDF Schema
SPARQL	Symantec Protocol and RDF Query Language
SIMD	Single Instruction Multiple Data
TSDB	Time series database
VByte	Variable-Byte

EXECUTIVE SUMMARY

The BigDataGrapes (BDG) platform aspires to provide components that go beyond the state-of-the-art on various stages of the management, processing, and usage of grapevine-related big data assets thus making easier for grapevine-powered industries to take important business decisions. The platform employs the necessary components for carrying out rigorous analytics processes on complex and heterogeneous data helping companies and organizations in the sector to evolve methods, standards and processes based on insights extracted from their data.

The goal of the BDG Distributed Indexing activity is to develop novel methodologies and components for realizing efficient indexing over distributed big data batch and cross-streaming sources.

The activities carried out in this first period focused on the design of time and space efficient indexing data structures for structured and unstructured data such as labelled trees, graphs, and text documents, including compression techniques for Big data management that support a broad range of analytical queries over arbitrary data dimensions. Specifically, we investigated the efficiency and effectiveness dimensions of indexes for RDF triples based on inverted indexes, and designed a novel compression technique for making these indexes more efficient in both space and time. This deliverable includes the first version of the software components developed and discusses the preliminary results obtained. An appendix shows how to access the software, install it and reproduce the tests conducted.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

This accompanying document for deliverable D3.3 (Distributed Indexing Components) reports about the work done and the software components implemented within Task 3.3 (Big Data Indexing) of WP3 (Data & Semantics Layer) of the BigDataGrapes (BDG) project. The goal of Task 3.3 is to develop novel methodologies and components for realizing efficient indexing over distributed big data batch and cross-streaming sources.

Specifically, the activities carried out in this first period focused on the design of time and space efficient data structures for indexing huge amount of structured and unstructured data such as labelled trees, graphs, and text documents, supporting a broad range of analytical queries over arbitrary data dimensions. This deliverable includes the first version of the software components developed and discusses the preliminary results obtained. In particular, we present a **novel compression technique** (Pibiri & Venturini 2018) for inverted indexes based on Variable-Byte, a well-known and widely adopted method **for coding integer sequences by saving memory space and enabling fast search operations**. As detailed in Section 2, the use of the inverted indexes is a common approach used to index RDF datasets. In this document we provide preliminary results obtained by applying the novel method on a large inverted index for RDF data.

The inverted index is the core data structure at the basis of search engines, massive database architectures and social networks. It is also one of the main solution used to index RDF datasets, for example Semplore (L. Zhang et al. 2007) and Siren (Delbru et al. 2010) use inverted indexes. In its simplicity, the inverted index can be regarded as being a collection of sorted integer sequences, called inverted or posting lists.

1.1 INVERTED INDEXES PRELIMINARIES

Given a collection D of documents, each document is identified by a non-negative integer called a document identifier, or docid. A posting list is associated to each term appearing in the collection, containing the list of the docids of all the documents in which the term occurs. The collection of the posting lists for all the terms is called the inverted index of D, while the set of the terms is usually referred to as the dictionary. Posting lists typically contain additional information about each document, such as the number of occurrences of the term in the document, and the set of positions where the term occurs.

Inverted index compression is essential to make efficient use of the memory hierarchy, thus maximizing query processing speed. Representing sequences of integers in compressed space is thus a fundamental problem, studied since the 1950s with applications going beyond inverted indexes.

A classical solution is based on sorting in increasing order each posting list and representing the sequence using the differences between consecutive numbers (d-gaps). Since the d-gaps are all positive numbers that can be encoded with uniquely- decodable variable length binary codes. Smaller the d-gaps less the average number of bits needed for their encoding.

It is fundamental for a Big Data management system to provide high throughput and, at the same time, return fast query answers to users. Clearly, a single search server with a single inverted index could be not sufficient to deal with such constraints. Therefore, the query processing subsystem is usually deployed on a cluster of servers which can adopt a replicated and/or distributed architecture.

In the replicated architecture, each cluster's server holds a replica of the same inverted index. Servers operate in parallel, processing different queries at the same time hence increasing the search engine throughput. When a user issues a query, it is first received by a broker, which routes the query on one search server. Once the search server has computed the query results, these are sent back to the issuing user. However, such replicated architecture does not have effects on query latency. From the user perspective, query latency is the amount of time elapsing between issuing the query and receiving its result. One way to reduce latencies is to reduce the query processing times. To this end, the index can be partitioned into smaller shards. In fact, query processing times increase with the posting lists' lengths, since more postings need to be traversed, decompressed, and scored. Therefore, index partitioning aims at keeping the posting lists short so that query processing times are reduced. For instance, document-based partitioning assign different documents to different shards, such that each shard can act as an independent inverted index. After partitioning, index shards are assigned to different search servers and incoming queries are dispatched to all search servers. Each server computes the query results on its shard independently from the others. These partial results are then aggregated and sent back to the issuing user.

As follows from the above discussion, the distribution and replication of indexes is a orthogonal dimension with respect to the choice of the data structure for storing and accessing the inverted index. Any implementation of an inverted index can be used in a replicated and distributed architecture designed, dimensioned and tuned to meet the given throughput and latency requirements. Anyway, due to the huge quantity of data available and processed on a daily basis by the mentioned systems, compressing the inverted index is indispensable since it can introduce a two-fold advantage over a non-compressed representation: feed faster memory levels with more data and, hence, speed up the query processing algorithms. As a result, the design of algorithms that compress the index effectively while maintaining a noticeable decoding speed is an old problem in computer science, that dates back to more than 50 years ago, and still a very active field of research. Many representations for inverted lists are known, each exposing a different compression ratio vs. query processing speed trade-off.

We point the reader to Section 2 for a concise overview of the different encoders that have been proposed through the years.

Among these, Variable-Byte (henceforth, VByte) is the most popular and used byte-aligned code. In particular, VByte owes its popularity to its sequential decoding speed and, indeed, it is the fastest representation up to date for integer sequences. For this reason, it is widely adopted by well-known companies as a key database design technology to enable fast search of records.

We mention some noticeable examples. Google uses VByte extensively: for compressing the posting lists of inverted indexes and as a binary wire format for its protocol buffers. IBM DB2 employs VByte to store the differences between successive record identifiers. Amazon patented an encoding scheme, based on VByte and called Varint-G8IU, which uses SIMD (Single Instruction Multiple Data) instructions to perform decoding faster. Many other storage architectures rely on VByte to support fast full-text search, like Redis, UpscaleDB and Dropbox.

We now quickly review how the VByte encoding works. The binary representation of a non-negative integer is divided into groups of 7 bits which are represented as a sequence of bytes. In particular, the 7 least significant bits of each byte are reserved for the data whereas the most significant (the 8-th), called the continuation bit, is equal to 1 to signal continuation of the byte sequence. The last byte of the sequence has its 8-th bit set to 0 to signal, instead, the termination of the byte sequence. Decoding is simple: we just need to read one byte at a time until we find a value smaller than 2^7.

The main drawback of VByte lies in its byte-aligned nature, which means that the number of bits needed to encode an integer cannot be less than 8. For this reason, VByte is only suitable for large numbers. However, the inverted lists are notably known to exhibit a clustering effect, i.e., these present regions of close identifiers that are far more compressible than highly scattered regions (Ottaviano & Venturini 2014). Such natural clusters are present because the indexed data itself tend to be very similar. The key point is that efficient inverted index compression should exploit as much as possible the clustering effect of the inverted lists. VByte currently fails to do so and, as a consequence, it is believed to be space-inefficient for inverted indexes.

Our paper (Pibiri & Venturini 2018) disproves the folklore belief that VByte is too large to be considered space-efficient for compressing inverted indexes. This is done by presenting Opt-VByte, an optimized VByte-based algorithm that improves compression ratio of VByte by a factor 2 on the standard Web pages. Although the literature reports about several index representations that outperform both in time and space VByte, one of the reason for being interested in improving VByte is that VByte is extremely popular and several existing systems use it. As our solution only introduces an optimization algorithm to run at construction time and the compression algorithm is essentially Vbyte, it can be adopted by any of these systems with a very small effort. This can thus have a large impact.

The basic idea is based on partitioning the inverted lists into blocks and representing each block with the most suitable encoder, chosen among VByte and the characteristic bit-vector representation. Partitioning the lists has the potential of adapting to the distribution of the integers in the lists by adopting VByte for the sparse regions where larger d-gaps are likely to be present.

Since we cannot expect the dense regions of the lists be always aligned with uniform boundaries, we consider the optimization problem of minimizing the space of representation of an inverted list by representing it with variable-length partitions. To solve the problem efficiently, we introduce an algorithm that finds the optimal partitioning in linear time and constant space.

1.2 ORGANIZATION OF THE DOCUMENT

The deliverable is organized as follows. Section 2 presents the state-of-the-art for inverted index representation, while Section 3 discusses the results of the experiments conducted to test our index on a large RDF dataset and experimentally comparing the efficiency of OPT-VByte against other competitors. We emphasize that OPT-VByte has been designed and implemented in the context of the BDG project. A paper discussing the advantages of OPT-VByte with respect to the state of the art is currently under revision for publication in a first-tier international journal. Experiments conducted to assess this technique for indexing RDF data show that OPT-VByte is better than any other approach but Partitioned Elias Fano (PEF) (Ottaviano & Venturini 2014), the state-of-the-art technique for coding inverted indexes developed by the same authors of OPT-VByte before the beginning of the BDG project. Section 4 complete the review of the state of the art by presenting the most popular tools for graph and time series indexing. These tools are widely used and their discussion provide a complementary view on the problem of efficient distributed indexing with respect to the research results previously discussed. We conclude with a plan for future work. Finally, an Appendix provides instructions to download the software, install it and test our index on the provided RDF dataset.

2 STATE OF THE ART

In this Section we describe the main approaches for indexing RDF data: Indexing based on Database Management systems, on B-tree, and on inverted indexes. As our implementation follows the latter approach, the remaining of the Section present a detailed overview of the main techniques for representing inverted indexes. This allows to fully understand our main contribute that will be described in the subsequent sections.

2.1 RDF INDEXING

Three different main approaches have been proposed to deal with the problem of solving RDF patterns, i.e., triples or quadruples. Database Management based systems manage RDF triples or quads by relying on existing RDBMS systems. Conversely, RDF-native systems are specifically designed to deal with RDF datasets. The index contains the whole dataset and has to provide very basic operations on it. These operations should efficiently solve the possible instances of any SPARQL pattern. The whole SPARQL query is then solved by joining the partial results of its patterns. A native index structure may consist of three B-trees (or its variants like B+-tree). Each of them stores the triples in the dataset indexed, respectively, by subject, predicate and object. Access patterns that contain two variables are solved trivially by querying the correct B-tree while access patterns with fewer than two variables ask for a join of partial result to obtain the final answer. Observe that the join may be computationally very expensive since it may be degenerated to a complete scan of the whole dataset. These poor worst-case guarantees lead researchers to design more efficient solutions. A possible solution consists on resorting to six different B-Trees. YARS2 (Harth & Decker 2005; Harth et al. 2007) reduces significantly the number of required B-trees. RDF-3X (Neumann & Weikum 2010) is currently among the best indexes. It uses six B+-trees for index triples, namely, it does not resort to the reduction proposed in (Harth & Decker 2005). Inverted-index based systems implement RDF-systems over inverted lists, the logical data structure generally used for speeding-up search in large repositories. Semplore (L. Zhang et al. 2007) is an index over semantic data that supports hybrid searches which integrate structured query parts with keyword context. Siren (Delbru et al. 2010) is a system based on a node indexing scheme. In their approach each element of any posting list is a path on a tree representation of the dataset. These solutions have a much lower space usage compared to the other approaches as they do not need any replication of the dataset. Furthermore, as we will see in the next subsection, compression of inverted lists is a mature field of research with several very effective solutions.

2.2 INVERTED LIST COMPRESSION

In this subsection we overview the most important compressors devised for efficient inverted list representation. Additionally, to the ones we review in the following, we remark that well-known compressors like Elias' Gamma and Delta and Golomb are known to obtain inferior compression ratios for inverted index storage with respect to the state-of-the-art, thus we do not consider them. The reader can refer to the paper (Pibiri & Venturini 2018) for a complete list of references.

2.2.1 Block-based

Given an increasingly ordered posting list representing the docid containing a given term, blocks of integers can be encoded separately, to improve both compression ratio and retrieval efficiency. This line of work finds its origin in the so-called Frame-of-reference.

A simple example of this approach, called binary packing, encodes blocks of fixed length, e.g., 128 integers. To reduce the value of the integers, we can subtract from integer the previous one (the first integer is left as it is), making each block be formed by integers greater than zero known as *delta*-gaps (or just d-gaps). Scanning a block will need to re-compute the original integers by computing the prefix sums.

In order to avoid the prefix sums, we can just encode the difference between the integers and the first element of the block (base+offset encoding).
Using more than one compressor to represent the blocks, rather than only one, can also introduce significant improvements in query time within the same space constraints.

Other binary packing strategies are Simple9, Simple8b, Simple16, and QMX, that combine relatively good compression ratio and high decompression speed. The key idea is to try to pack as many integers as possible in a memory register (32, 64 or 128 bits). Along with the data bits, a *selector* is used to indicate how many integers have been packed together in a single unit. In the QMX mechanism the selectors are run-length encoded.

2.2.2 PForDelta

The biggest limitation of block-based strategies is that these are inefficient whenever a block contains at least one large element, because this causes the compressor to use a number of bits per element proportional to the one needed to represent that large value. To overcome this limitation, PForDelta was proposed. The main idea is to choose a proper value k for the universe of representation of the block, such that a large fraction, e.g., 90%, of its integers fall in the range $[b, b + 2^k - 1]$ and, thus, can be written with k bits each. This strategy is called patching. All integers that do not fit in k bits, are treated as exceptions and encoded separately using another compressor.

The optimized variant of the encoding (Opt-PFOR), which selects for each block the values of b and k that minimize its space occupancy, has been demonstrated to be more space-efficient and only slightly slower than the original PForDelta.

2.2.3 Elias-Fano

This strategy directly encodes a monotone integer sequence without a first delta encoding step. It was independently proposed by Elias and Fano, hence its name. Given a sequence of size n and universe u, its Elias-Fano representation takes at most n log u/n + 2n bits, which can be shown to be less than half a bit away from the information-theoretic lower bound. The encoding has been recently applied to the representation of inverted indexes and social networks, thanks to its excellent space efficiency and powerful search capabilities, namely random access in O(1) and successor queries in O(1 + log u/n) time. The latter operation which, given an integer x of a sequence S returns the smallest integer y in S such that y <= x, is the fundamental one when resolving boolean conjunctions over inverted lists. If you pick a random sequence of n numbers up to m, then Elias-Fano is (almost) optimal. However, real-world inverted lists are far from being random sequences as they have clusters of consecutive (or almost consecutive) integers.

The partitioned variant of Elias-Fano (PEF) (Ottaviano & Venturini 2014), splits a sequence into variable-sized partitions and represents each partition with Elias-Fano. The partitioned representation sensibly improves the compression ratio of Elias-Fano by preserving its query processing speed. In particular, it currently embodies the best trade-off between index space and query processing speed.

2.2.4 Binary Interpolative Coding

Binary Interpolative Coding (BIC) (Moffat & Stuiver 2000) is another approach that, like Elias-Fano, directly compresses a monotonically increasing integer sequence. In short, BIC is a recursive algorithm that first encodes the middle element of the current range and then applies this encoding step to both halves. At each step of recursion, the algorithm knows the reduced ranges that will be used to write the middle elements in fewer bits during the next recursive calls.

Many papers in the literature experimentally proved that BIC is one of the most space-efficient method for storing highly clustered sequences, though among the slowest at performing decoding (Ottaviano & Venturini 2014).

2.2.5 The Variable-Byte family

Various encoding formats for VB have been proposed in the literature in order to improve its sequential decoding speed. By assuming that the largest represented integer fits into 4 bytes, two bits are sufficient to describe the proper number of bytes needed to represent an integer. In this way, groups of four integers require one control byte that must be read once as a header information. This optimization was introduced in Google's Varint-GB and reduces the probability of a branch misprediction which, in turn, leads to higher instruction throughput. Working with byte-aligned codes also opens the possibility of exploiting the parallelism of SIMD instructions of modern processors to further enhance the decoding speed. This is the line of research taken by the recent proposals that we overview below.

Varint-G8IU uses a similar idea to the one of Varint-GB but it fixes the number of compressed bytes rather than the number of integers: one control byte is used to describe a variable number of integers in a data segment of exactly 8 bytes, therefore each group can contain between two and eight compressed integers.

Masked-VByte directly works on the original VB format. The decoder first gathers the most significant bits of consecutive bytes using a dedicated SIMD instruction. Then, using previously-built look-up tables and a shuffle instruction, the data bytes are permuted to obtain the original integers.

Stream-VByte, instead, separates the encoding of the control bytes from the data bytes, by writing them into separate streams. This organization permits to decode multiple control bytes simultaneously and, therefore, reduce branch mispredictions that can stop the CPU pipeline execution when decoding the data stream.

3 BIGDATAGRAPES RDF INDEXING

3.1 EXPERIMENTS

We report here the result of an experiment we performed to test the efficiency of different index representations. We measure the efficiency of an index representation with respect three main characteristics: index construction time, index space usage, and query time.

The experiment uses an RDF dataset obtained from DBpedia. We removed all the inverted lists shorter than 128 postings. These very short lists can be treated in a different and more efficient way (e.g., no compression). The resulting dataset has more than two Billion postings.

All the experiments were run on a machine with Intel i7 -4790K CPU with 4 cores (8 threads) clocked at 4.00GHz and with 32GB of RAM DDR3, running Linux 4.13.0 (Ubuntu 17.10), 64 bits.

To test the building time of the indexes we measure the time needed to perform the whole building process, that is: (1) fetch the posting lists from disk to main memory; (2) encode them in main memory; (3) save the whole index data structure back to a file on disk.

Since the process is mostly I/O bound, we make sure to avoid disk caching effects by clearing the disk cache before building the indexes.

To test the query processing speed of the indexes, we memory map the index data structures on disk and compute boolean conjunctions over a set of random queries drawn. Each query specifies the elements of a triple and searches for all the triple matching the unspecified one. We used 600,000 queries.

We repeat each experiment three times to smooth fluctuations in the measurements and consider the mean value. The query algorithm runs on a single core and timings are reported in microseconds.

The results are reported in the Figure 1 below. The table compares most of the algorithm presented in Section 2 against our novel proposal: Opt-VByte (Pibiri & Venturini 2018) with respect to: building time, space usage and query time.

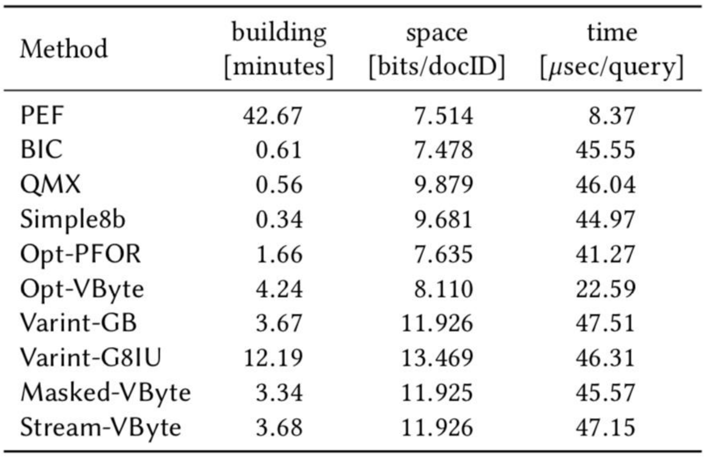
Even if Opt-VByte optimally partitions each inverted list before compressing it, its building time is very competitive, being very close to the one of non-optimized compressors (e.g., Varint-GB and other VByte method).

Opt-VByte is better than any other VByte-based approach (Varint-*, Masked-VByte and Stream-VByte) wrt to space usage and query time. Indeed, it improves the space usage by a factor more than 1.4 and the query time by a factor more than 2. This makes it the best VByte approach with margin.

Space usage of Opt-VByte is also very close to the one of the best compressors (e.g., BIC uses less than 0.5 bits per posting less than Opt-VByte).

Query time of Opt-VByte is 2 times faster than any other competitor but PEF. PEF instead is much faster (a factor 2.8 faster than Opt-VByte).

We observe that PEF is both faster and smaller than Opt-VByte. Thus, PEF results as the best solution if the building time is not a main concern. However, the better query time of PEF wrt Opt-VByte is quite surprisingly and it may be due to the properties of the query set we used. Indeed, queries are very selective (i.e., they return a very small number of results). In this setting the base algorithm of PEF (i.e., Elias-Fano) is much more efficient than the base algorithm of Opt-VByte (i.e., VByte). However, the introduction of a real set of queries may change this aspect. Thus, the plan for the future is to: 1) repeat the experiment with a real set of queries and other datasets; 2) try to combine PEF and Opt-VByte to get the best of the two for RDF indexing.



Method	building [minutes]	space [bits/docID]	time [μsec/query]
PEF	42.67	7.514	8.37
BIC	0.61	7.478	45.55
QMX	0.56	9.879	46.04
Simple8b	0.34	9.681	44.97
Opt-PFOR	1.66	7.635	41.27
Opt-VByte	4.24	8.110	22.59
Varint-GB	3.67	11.926	47.51
Varint-G8IU	12.19	13.469	46.31
Masked-VByte	3.34	11.925	45.57
Stream-VByte	3.68	11.926	47.15

Figure 1: The performance of various compressors for the DBPedia dataset, expressed as: time for building the indexes (in minutes), space (bits per docID) and query time (μsec per query)

4 STATE-OF-THE-ART OF TECHNOLOGICAL TOOLS

4.1 TOOLS FOR GRAPH-BASED INDEXING

Graph databases – and consequently triple stores – show their power with respect to conventional storage and indexing schemes as queries become more complex or follow relations that are deeper than first level. A graph database doesn't utilize foreign keys or JOIN operations. Instead, all relationships are natively stored within vertices. This results in deep traversal capabilities, increased flexibility and enhanced agility.

Graph databases are consequently equipped to easily accommodate rapidly scaling data and easily expand the underlying schema describing the data. Several graph databases solutions have been proposed and are being distributed, often in the context of a general data management environment.

Neo4j[1] is the most popular graph database system[2] at the time of writing. It is a native graph storage framework, following the property graph model for representing and storing data, i.e. the representation model conceptualises information as nodes, edges or properties. Accessing and querying the underlying data is achieved via the usage of the open-sourced Cypher query language, originally developed exclusively for Neo4j.

Titan[3] is a GDBMS optimised for the management of large-scale graphs distributed across multi-machine clusters of arbitrary size. Titan is not a native graph store, instead supporting different backends like Apache Cassandra and Oracles' BerkeleyDB. Its search mechanism provides connectors to popular enterprise search platforms like Solr and Elasticsearch.

Cayley[4] is based on the graph backend of Freebase and Google Knowledge Graph. It also isn't a native graph storage system, as it relies on key-value pairs and traditional relational databases for storing and indexing.

GraphDB[5] is an RDF triplestore compliant with the core semantic web W3C specifications (RDF, RDFS, OWL). It acts as a SAIL over the RDF4J framework[6], thus providing functionalities for all critical semantic graph operations (storing, indexing, reasoning, querying, etc.). The query language used is the implementation of the SPARQL 1.1 specifications, while connectors with Elasticsearch and Lucence are incorporated in the system.

AllegroGraph[7] is also a native graph database following the core semantic web standards. While it is closed-source and generally relies on its own implementation for storage and indexing, it also provides integration with full-text search frameworks and standardized languages for querying (SPARQL and Prolog).

OrientDB[8] follows the Property Graph model to actually handle different types of data, abstracting their representation via the usage of an application-specific Object Data Model. Accordingly, it support different indexing mechanisms, relying on Lucene for full-text and spatial indexing.

[1] https://neo4j.com

[2] https://db-engines.com/en/ranking/graph+dbms

[3] http://titan.thinkaurelius.com

[4] https://cayley.io

[5] http://graphdb.ontotext.com

[6] http://rdf4j.org

[7] https://franz.com/agraph/allegrograph/

[8] https://orientdb.com

In addition to pure graph databases, several multi-model management systems have incorporated the management of graph data structures to their functionality. A brief overview of the most prominent such systems follows.

MarkLogic[9] is a multi-model DBMS, initially conceived as a document-based NoSQL platform, but adding support for the management of semantic data expressed in RDF.

Virtuoso[10] is an engine that acts as a single-point server and middleware for multiple data management paradigms (relational, object-relational, XML, RDF, file-based). The underlying storage mechanism is a traditional relational database with abstraction and serialisation components built into the framework for exposing data of the aforementioned different representations. RDF data are accessed and queried using an extension of the SPARQL specification.

ArangoDB[11] is a document-based NoSQL DBMS that support graph data representation via a generic vertex-edge model. The data are actually stored in a JSON-based binary format and queried through AQL, a custom query language. ArangoDB comprises multiple indexing mechanisms, among them a vertex-centric index optimized for handling graphs.

4.2 TOOLS FOR TIME SERIES INDEXING

Although, time series data can be stored in traditional relational databases, in case of real-time applications the high data volumes as well as the large number of transactions makes their use not practical.

That is why is necessary a system that is built especially for handling metrics and events or measurements that are time-stamped. A time series database (TSDB) is optimized to handle such data by creating indices based on a timestamp or a time range.

Moreover, a TSDB provides the facilities to create, manage and organize time series, along with basic calculations over the series. These calculations include, multiplying, adding or combining time series to form new time series.

The most popular time series databases are InfluxDB[12] and Elasticsearch[13]. InfluxDB is especially optimized for storing and querying data points with a timestamp. It stores the data following the Log-structured merge-tree (LSM) tree paradigm [14] and supports data transformation and selection queries, through client libraries and REST APIs.

On the other hand, Elasticsearch is a distributed database, providing a full-text search engine based on Lucene[15]. The distributed nature of Elasticsearch, allows near real-time search in all kinds of documents. The indices of Elasticsearch can be divided into shards, hence supporting automatic rebalancing and routing. Moreover, the indices can be replicated to support efficient fault-tolerance.

Furthermore, Elasticsearch encapsulates out-of-the-box methods for establishing connections with messaging systems like Kafka, which makes integration easier and allows the faster development of real-time applications.

[9] https://www.marklogic.com

[10] https://virtuoso.openlinksw.com

[11] https://www.arangodb.com

[12] https://www.influxdata.com/

[13]https://www.elastic.co/

[14] O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). Acta Informatica, 33(4), 351-385.

[15]http://lucene.apache.org/

Contrary to InfluxDB, the first step to store any data to Elasticsearch is to define how the data are mapped. This enables the use of the advanced aggregation/ facets mechanism, which allows the building of advanced and complex queries targeting the actual content of the timestamped documents.

The smooth integration of Elasticsearch with messaging systems and the fact that Elasticsearch encapsulates with and advanced search engines, makes it a fine candidate for the purposes of the BDG project.

5 SUMMARY

The activities carried out in T3.3 during this first period focused on the design of time and space efficient data structures for indexing complex data that can support a broad range of analytical queries over arbitrary data dimensions. This deliverable presented the first version of the software components implementing a novel compression technique for inverted indexes based on Variable-Byte, a well-known and widely adopted method for coding integer sequences by saving memory space and enabling fast search operations. The design and implementation of this novel indexing technique have been detailed in a scientific paper currently under revision for publication (Pibiri & Venturini 2018). This accompanying document introduced the context for understanding our contribution to the advance of the state of the art in the field, and report about the experiments conducted to assess the efficiency of the method for indexing and searching large RDF datasets. Furthermore, it discussed the main tools available to index and manage graph and time series. These tools are very popular and have been successfully used in a plenty of applications.

6 REFERENCES

Delbru, R. et al., 2010. A Node Indexing Scheme for Web Entity Retrieval. In L. Aroyo et al., eds. *The Semantic Web: Research and Applications*. Springer Berlin / Heidelberg, pp. 240-256. Available at: http://dx.doi.org/10.1007/978-3-642-13489-0_17.

Harth, A. & Decker, S., 2005. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the Third Latin American Web Congress*. Washington, DC, USA: IEEE Computer Society, p. 71--. Available at: http://dl.acm.org/citation.cfm?id=1114687.1114857.

Harth, A. et al., 2007. YARS2: a federated repository for querying graph structured data from the web. In *Proceedings of the 6th international, The semantic web and 2nd Asian conference on Asian semantic web conference*. Berlin, Heidelberg: Springer-Verlag, pp. 211-224. Available at: http://dl.acm.org/citation.cfm?id=1785162.1785179.

Moffat A. & Stuiver L., 2000. Binary interpolative coding for effective index compression. Information Retrieval Journal, 3(1):25–47.

Neumann, T. & Weikum, G., 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), pp.91-113. Available at: http://dx.doi.org/10.1007/s00778-009-0165-y.

Pibiri, G. & Venturini, R., 2018. Variable-Byte encoding is now space-efficient too. Available at: https://arxiv.org/abs/1804.10949

Ottaviano, G. & Venturini, R., 2014. Partitioned Elias-Fano indexes. In Proceedings of the 37th International Conference on Research and Development in Information Retrieval (SIGIR), pages 273–282.

Zhang, L. et al., 2007. Semplore: an IR approach to scalable hybrid query of semantic web data. In *Proceedings of the 6th international, The semantic web and 2nd Asian conference on Asian semantic web conference*. Berlin, Heidelberg: Springer-Verlag, pp. 652-665. Available at: http://dl.acm.org/citation.cfm?id=1785162.1785210.

7 APPENDIX

7.1 SETUP & INSTALL THE RDF INDEX

7.1.1 Getting the code

To get the code, clone it from GitHub

```
git clone https://github.com/jermp/opt_vbyte
```

Currently, this is a private repository as the paper is still under submission. We will release the code upon the publication of the paper.

7.1.2 Building the code

The code is tested on Linux Ubuntu with gcc 7.3.0. The following dependencies are needed for the build: CMake >= 2.8 and Boost >= 1.42.0.
The code is largely based on the ds2i project, so it depends on several submodules. If you have cloned the repository without --recursive, you will need to perform the following commands before building:

```
$ git submodule init
$ git submodule update
```

To build the code on Unix systems (see file CMakeLists.txt for the used compilation flags), it is sufficient to do the following:

```
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make -j[number of jobs]
```

Setting [number of jobs] is recommended, e.g., make -j4.

Unless otherwise specified, for the rest of this guide we assume that we type the terminal commands of the following examples from the created directory build.

7.1.3 Input data format

The collection containing the docID and frequency lists follow the format of ds2i, that is all integer lists are prefixed by their length written as 32-bit little-endian unsigned integers:

- <basename>.docs starts with a singleton binary sequence where its only integer is the number of documents in the collection. It is then followed by one binary sequence for each posting list, in order of term-ids. Each posting list contains the sequence of docIDs containing the term.
- <basename>.freqs is composed of a one binary sequence per posting list, where each sequence contains the occurrence counts of the postings, aligned with the previous file (note however that this file does not have an additional singleton list at its beginning).

The data subfolder contains an example of such collection organization, for a total of 113,306 sequences and 3,327,520 postings. The queries file is, instead, a collection of 500 (multi-term) queries. For the following examples, we assume to work with the sample data contained in data.

7.1.4 FROM RDF Input data format

We can use the script rdf_to_index.sh to convert a RDF dataset (in .ttl format) to the input data format described above. The script, given a ttl file, processes it and produces an inverted list for each subject, predicate and object. These inverted lists contain identifiers of all the triples that contain that subject, predicate and object. The script also produces a random set of queries that can be used to test the efficiency of the different indexes.

7.1.5 Building the indexes

The executables src/create_freq_index should be used to build the indexes, given an input collection. To know the parameters needed by the executable, just type

```
$ ./create_freq_index
```

without any parameters. You will get:

```
$ Usage ./create_freq_index:
$      <index_type> <collection_basename> [--out <output_filename>] [--F <fix_cost>] [--check]
```

Below we show some examples.

Example 1.

The command

```
$ ./create_freq_index opt_vb ../data/test_collection --out test.opt_vb.bin
```

builds an optimally-partitioned VByte index that is serialized to the binary file test.opt_vb.bin.

Example 2.

The command

```
$ ./create_freq_index block_maskedvbyte ../data/test_collection --out test.vb.bin
```

builds an un-partitioned VByte index that is serialized to the binary file test.vb.bin, using Macked-VByte to perform sequential decoding.

Example 3.

The command

```
$ ./queries opt_vb and test.opt_vb.bin ../data/queries
```

performs the boolean AND queries contained in the data file queries over the index serialized to test.opt_vb.bin.

NOTE: See also the Python scripts in the scripts/ directory to build the indexes and collect query timings.

7.2 ELASTICSEARCH DOCUMENTATION & TOOLS

Documentation/ Tool	Description	Link
Elasticsearch	Official Documentation	https://www.elastic.co/guide/index.html
Kibana	Development & Dashboard Workbench	https://www.elastic.co/products/kibana
Configuration Model	Tool that estimates the configuration of Elasticsearch, given various scenarios	https://docs.google.com/spreadsheets/d/1r5HmOlv6dfN_EVe8Pyfx3GEl16LbALPPQ2Geym6bgx8/edit?usp=sharing
Docker	Dockerized versions of Elasticsearch & Kibana	https://www.docker.elastic.co/