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

D3.3 - Distributed Indexing Components

DELIVERABLE NUMBER	D3.3
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	Nikos Manouselis
ADDRESS	110 Pentelis Str., Marousi, GR15126, Greece
REPLY TO	nikosm@agroknowc.com
PHONE	+30 210 6897 905
EU PROJECT OFFICER	Ms. Annamária Nagy
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 June 2020 (M30)
DATE OF DELIVERY (SUBMITTED)	31 July 2020 (M31)
VERSION | STATUS	3.0
NATURE	Demonstrator (DEM)
DISSEMINATION LEVEL	Public (PU)
AUTHORS (PARTNER)	Rossano Venturini (CNR), Giulio Ermanno Pibiri (CNR), Raffaele Perego (CNR)
CONTRIBUTORS	Milena Yankova (ONTOTEXT), Pythagoras Karampiperis (Agroknow), 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, Raffaele Perego (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)
1.1	Update of Deliverable at M21	18/09/2019	Giulio Ermanno Pibiri (CNR), Raffaele Perego (CNR)
1.2	Input from partners	20/9/2019	Nikola Rusinov (ONTOTEXT)
1.1	Internal Review	23/09/2019	Nikola Tulechki (ONTOTEXT)
2.0	Final edits after internal review	23/09/2019	Raffaele Perego (CNR)

2.1	Update of Deliverable at M30	25/06/2020	Giulio Ermanno Pibiri, Raffaele Perego, Franco Maria Nardini (CNR)
2.2	Internal review	23/06/2020	Florian Schlenz (Geocledian)
3.0	Final edits after internal review	30/06/2020	Raffaele Perego (CNR)

PARTICIPANTS		CONTACT
Agroknow IKE (Agroknow, Greece)		Nikos Manouselis Email: nikosm@agroknowc.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
SYMBEEOSIS EY ZHN S.A. (Symbeeosis, Greece)		Konstantinos Rodopoulos Email: rodopoulos-k@symbeeosis.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 Elias 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
3T	Three-trie index
2T	Two-trie index

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, heterogeneous big data batch and cross-streaming sources.

The activities carried out in this activity focus on the design of time and space efficient indexing data structures for structured and unstructured data such as RDF graphs, time series and text documents, including compression techniques for Big data management that support a broad range of analytical queries over arbitrary data dimensions.

Specifically, we investigate the efficiency and effectiveness dimensions of indexes for texts, RDF triples, and time-series. This investigation led us to develop two novel techniques based on inverted indexes and succinct tries. These solutions substantially outperform competitive approaches at the state-of-the-art. Both these scientific results have been already published in Transactions on Knowledge and Data Engineering (TKDE) (Pibiri & Venturini 2019b; Perego, Pibiri & Venturini 2020) — a top-tier journal in Computer Science.

We believe that the above project results will have a high impact for both the project partners and the scientific community working in the field.

A third contribution on indexing time-series has been added to the last update of the deliverable due at M30. The software solution discussed is still under development but the encouraging results achieved so far lead us to believe that the solution designed will be very useful and impactful for the project and the community. Moreover, a top-tier publication is planned even for this project result.

This deliverable describes the software components developed and discusses the promising results obtained. An appendix shows how to access the software, install it and reproduce the tests conducted.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 period focused on the design of time- and space-efficient data structures for indexing large amounts of structured and unstructured data such as RDF graphs and text documents, supporting a broad range of analytical queries over arbitrary data dimensions. This deliverable includes the software components developed and discusses the obtained results measured on huge datasets and different testing conditions. Our contribution is three-fold.

First, we present a novel compression technique (Pibiri & Venturini 2019b) 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.

Second, we design a family of compressed indexes (Perego, Pibiri & Venturini 2020) based on the trie data structure and show that this novel index can support any kind of RDF query pattern involving 0, 1, 2 or 3 wild card symbols.

Third, we sketch a novel compression/decompression algorithm for time-series that, while still under development and tuning, achieves a significant improvement over the state-of-the-art.

As detailed in Section 2, inverted indexes and tries are the best common approaches used to index text and large RDF datasets, respectively. In this document we illustrate our techniques and provide the results of the experiments conducted to assess the performance of the devised approaches.

1.1. ORGANIZATION OF THE DOCUMENT

The deliverable is organized as follows.

Section 2 presents the state-of-the-art for inverted index compression, existing solutions to index RDF data, and time-series, whereas **Section 3** introduces and discusses our novel solutions by also presenting some experimental results.

We emphasize that all the proposed solutions have been designed and implemented in the context of the BDG project. In particular, the approaches based on inverted indexes and tries have been already published in Transactions on Knowledge and Data Engineering (TKDE) (Pibiri & Venturini 2019b; Perego, Pibiri & Venturini 2020). The third work, concerning indexing of time-series, is still under development.

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 solutions on the provided RDF dataset.

2. STATE OF THE ART

In this section we describe the main approaches for indexing data based on inverted indexes, Database Management systems, B-trees, and tries. The aim of this section is that of giving the necessary background to fully understand our proposal that we detail in Section 3.

2.1. INVERTED INDEX COMPRESSION

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 solutions used to index RDF datasets (L. Zhang et al. 2007; Delbru et al. 2010). In its simplicity, the inverted index can be regarded as being a collection of sorted integer sequences, called inverted or posting lists.

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 an 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.

In the following subsections we describe some techniques suitable for effective index compression without the goal of being exhaustive. The interested reader can find a complete treatment of all techniques in the survey (Pibiri and Venturini, 2019a).

2.1.1. 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.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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).

2.1.5. The Variable-Byte family

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 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.

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.

2.2. 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.

A completely different approach relies on representing the triples in memory with bitmaps.
BitMat (Atre et al. 2010) encodes the RDF data with a 3D bit-cube, where the three dimensions correspond to S, P and O, respectively.
TripleBit (Yuan et al. 2013) is a recent RDF store that codes triples with a bit matrix. In this matrix each column represents a distinct triple where only two bits are set to 1 in correspondence of the rows associated with the subject and the object of the triple. Symmetrically, each row corresponds to a distinct entity value occurring as subject or object in the triples uniquely identified by the bits set to 1 in the row. Since the resulting bit matrix is

very sparse it is compressed at the column level by simply coding the position of the two rows corresponding to the bits set, i.e., the identifiers of the subject and object of the associated triple. The experimental assessment discussed in the paper shows that TripleBit outperforms RDF-3X and BitMat by up to 2 — 4 orders of magnitude on large RDF datasets. Nevertheless, the space occupancy and scalability of such techniques is not very good.

Other authors investigated how variations of well-known data structures like k^2-trees (Brisaboa et al. 2009) and the Sadakane's compressed suffix array (CSA) (Sadakane 2003) can be exploited to compactly represent RDF datasets. In particular, k^2-TRIPLES (Álvarez-García et al. 2015) partitions the dataset into disjoint sub- sets of *(subject, object)* pairs, one per predicate, and represents the (highly) sparse bit matrices with k^2-trees. Another approach called RDFCSA (Brisaboa et al. 2015) builds an integer CSA index (Farina et al. 2012) over the sequence of concatenated triple IDs, with the use of truncated Huffman codes on integer gaps and run lengths for optimized performance. The ex- perimental assessment shows that RDFCSA requires roughly twice the space of the k^2-TRIPLES but it is up to two order of magnitude faster than the former. Although both these solutions outperform existing solutions like RDF-3X in both space usage and query efficiency, no implementation is publicly available to reproduce their results (personal communication).

HDT-FoQ (Focused on Querying) (Martínez-Prieto et al. 2012) is a trie-based solution that exploits the skewed structure of RDF graphs to reduce space occupancy while supporting fast querying. The HDT-FoQ format includes a header, detailing logical and physical metadata, the dictionary, encoding all the unique entities occurring in the triples as integers, and the set of triples encoded in a single SPO trie data structure. In order to support predicate-based retrieval, the second level of the trie is represented with a wavelet tree (Grossi, Gupta & Vitter 2003). Finally, additional inverted lists are maintained for object-based triple retrieval. In particular, for each object *o*, an inverted list is built, listing all pairs *(subject, predicate)* of the triples that contain o. Thus, searches are carried out by accessing each pair and searching for it in the trie.

2.3. TIME-SERIES INDEXING

Gorilla is an in-memory time-series database developed at Facebook (Pelkonen 2015). The time instants T(n) are delta-encoded using a universal compressor, which works remarkably well because these instants are very close, thus their difference very small.

The compression of the values V(n) is more involved and inspired by works on lossless compression of floating point values, such as (Ratanaworabhan et al., 2006) and (Lindstrom and Isenburg, 2006).
The idea used to compress a floating point value x is that of predicting a value, say y, that is very close to the real one. In the two values are very close, then the sign, the exponent, and the first few mantissa bits will be the same. Taking the XOR between the binary representation of x and y will turn
equal bits into zeros. Since the sign, the exponent, and the top mantissa bits occupy the most significant bit positions in the IEEE 754 standard, we can expect the XOR result to have a substantial number of leading zeros. Hence, XOR(x,y) can be encoded by a leading-zero counter that is followed by the remaining bits.

Gorilla applies this idea but with no expensive prediction calculations, as the XOR is always computed between a value and the previous one, noting that — as it holds for T(n) — also V(n) exhibits locality and values that are close together are likely to be very similar.

3. RDF AND TEXT INDEXING IN BIGDATAGRAPES

In this section we present two novel solutions to the problem of indexing text and RDF data in small space and supporting fast retrieval operations.

The first solution, presented in Section 3.1, is based on an inverted index data structure. We develop a novel compression format to compactly represent inverted lists that is an optimisation of the well-known Variable-Byte (VByte) encoding. Our technique, named Opt-VByte, is detailed and discussed in a paper published in the journal IEEE Transactions on Knowledge and Data Engineering (Pibiri & Venturini 2019b). The interested readers should refer to it for all details.

The second solution, presented in Section 3.2, is based on trie data structures instead. We designed compressed trie-shaped indexes that materialize S-P-O triples permutations to allow efficient pattern matching operations involving all possible triple selection patterns. To learn more about the approach, the interested readers should refer to the paper Perego, Pibiri & Venturini 2020, again published by IEEE TKDE.

Both sections illustrate extensive experimental results assessing the performance of our approaches on datasets relevant for our project.

3.1. INVERTED INDEXING

In Section 2.1 we provided general background on inverted index compression and discussed the Variable-Byte (VByte) encoding technique in Section 2.5.1. Here we briefly describe a novel solution based on VByte, exhaustively discussed and assessed in (Pibiri & Venturini 2019b).

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. Such natural clusters are present because the indexed data itself tends 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 2019b) 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 that is an optimized VByte-based algorithm, improving the compression ratio of VByte by a factor of 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 reasons for being interested in improving VByte is that VByte is extremely popular and several existing systems use it.

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.

As our solution introduces an optimization algorithm that runs at construction time only (not at query time) and the compression algorithm is essentially VByte, it can be adopted by any of these systems with a very small

effort. Thus, this can have a large impact.

The basic idea is to partition the inverted lists into blocks and represent 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 integers in the list 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.

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)

3.1.1. Experiments

We report here the results 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. While in (Pibiri & Venturini 2019b) the assessment is conducted on publicly available datasets of text document commonly used in the community, here we discuss the assessment of the same technique on inverted files representing RDF datasets. Specifically, the experiment detailed here 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.

The experiment was 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 Figure 1. The table compares most of the algorithm presented in Section 2 against our novel proposal — Opt-VByte — taking into account the following performance characteristics: 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 of more than 1.4 and the query time by a factor of more than 2. This makes it the best VByte approach with margin.
Space usage of Opt-VByte is also very close to 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.

3.2. TRIE-BASED INDEXING

In this section, we focus on the RDF *triple indexing problem* that is designing a static index on the integer triples that attains to efficient resolution of all possible selection patterns using as little space as possible. This is crucial for guaranteeing practical SPARQL query evaluation.

Moving from a critical analysis of the state-of-the-art, we note that existing solutions to the problem require too much space, because these either: rely on materializing *all* possible permutations of the S-P-O components; use expensive additional supporting structures; do not use sophisticated data compression techniques to effectively reduce the space for encoding triple identifiers.
Furthermore, this additional space overhead does not always pay off in terms of reduced query response time. The aim of this section is that of addressing these issues by illustrating compressed indexes for RDF data that are both compact *and* fast.

For all the details and the full set of experiments conducted, please refer to our paper (Perego, Pibiri & Venturini 2020).

Base implementation. As a high-level overview, our index maintains three different permutations of the triples, with each permutation sorted to allow efficient searches and effective compression as we are going to detail next. The permutations chosen are SPO, POS and OSP in order to (symmetrically) support all the six different triple selection patterns with one or two wildcard symbols: SP? and S?? over SPO; ?PO and ?P? over POS; S?O and ??O over OSP. The two additional patterns with, respectively, all symbols specified or none, can be resolved over any permutation, e.g., over the canonical SPO in order to avoid permuting back each returned triple.
Each permutation of the triples is represented as a trie with 3 levels, where nodes at the same level concatenated together to form an integer sequence. We keep track of where groups of siblings begin and end in the concatenated sequence of nodes by storing such pointers as absolute positions in the sequence of nodes. There- fore, the pointers are integer sequences as well. Since the triples are represented by the trie data structure in sorted order, the n node IDs in the first level of each trie are always complete sequences of integers ranging from 0 to n − 1 and, thus, can be omitted.

The advantage of this layout is two-fold: first, we can effectively compress the integer sequences that constitute the levels of the tries to achieve small storage requirements; second, the triple selection patterns are made cache-friendly and, hence, efficient by requiring to simply scan ranges of consecutive nodes in the trie levels.

In the following, we refer to this solution as the 3T index.

Cross compression. The described index layout represents the triples three times in different (cyclic) permutations in order to optimally solve all triple selection patterns. However, the data structure does not take advantage of the fact that the same set of triples is represented multiple times and, consequently, the index has an abundance of redundant information. It is possible to employ levels of the tries to compress levels of other tries, thus holistically *cross-compressing* the different permutations.

Cross-compression works by noting this crucial property: the nodes belonging to the subtree rooted in the second level of trie j are a subset of the nodes belonging to the subtree rooted in the first level of trie i, with j = (i + 2) mod 3, for i = 0, 1, 2. The correctness of this property follows automatically by taking into account that the triples indexed by each permutation are the same. Therefore, the children of x in the second level of trie j can be rewritten as the positions they take in the (larger, enclosing) set of children of x in the first level of trie i. Re-writing the node IDs as positions relative to the set of children of a sub-tree yields a clear space optimization because the number of children of a given node is much smaller (on average) than the number of distinct subjects or objects. (See also Figure 2 for the precise statistics).

Eliminating a permutation. The low number of predicates exhibited by RDF data leads us to consider a different select algorithm for the resolution of the query pattern S?O, able to take advantage of such skewed distribution. The idea is to pattern match S?O directly over the SPO permutation. For a given subject *s* and object *o*, in short, we operate as follows. We consider the set of all the predicates that are children of *s*. For each predicate pi in the set, we determine if the object *o* is a child of p_i with a search operation: if it is, then (s, p_i, o) is a triple to return.
The correctness of the algorithm is immediate and we argue that its efficiency is due to the following facts.

The small number of predicates as children of a given subject implies that the algorithm will perform few iterations: while these children are few per se (for example, at most 52 for the DBpedia dataset), the iterations will be on average far less. For a given *(subject, predicate)* pair, we have a very limited number of children to be searched for o, thus making the search operation run faster by short scans rather than via binary search.

In the light of the just described algorithm, we consider another index layout. In fact, now five out of the eight different selection patterns can be solved efficiently by the trie SPO, i.e.: SPO, SP?, S??, S?O and ???. In order to support other two selection patterns, we can either choose to: (1) materialize the permutation POS for predicate- based retrieval (query patterns ?PO and ?P?); (2) materialize the permutation OPS for object-based retrieval (query patterns ?PO and ??O). The choice of which permutation to maintain depends on the statistics of the selection patterns that have to be supported. We stress that the introduced algorithm allows us to actually save the space for a third permutation that costs roughly 1/3 of the whole space of the index.

We call this solution the 2T index, with two concrete instantiations: 2Tp (predicate-based) and 2To (object-based).

Dataset	Triples	Subjects (S)	Predicates (P)	Objects (O)
DBLP	88,150,324	5,125,936	27	36,413,780
Geonames	123,020,821	8,345,450	26	42,728,317
DBpedia	351,592,624	27,318,781	1480	115,872,941
Freebase	2,067,068,154	102,001,451	770,415	438,832,462

Figure 2: Real-world RDF datasets statistics.

3.2.1. Experiments

We perform our experimental analysis on very large and publicly available datasets, whose statistics are summarized in Figure 2.
All the experiments are performed on a server machine with 4 Intel i7-7700 cores (@3.6 GHz), 64 GB of RAM DDR3 (@2.133 GHz) and running Linux 4.4.0, 64 bits. We compiled the code with gcc 7.3.0 using the highest optimization setting, i.e., with compilation flags -O3 and -march=native.
To measure the query processing speed, we use a set of 5000 triples drawn at random from the datasets and set 0, 1 or 2 wildcard symbols. In all tables, percentages and speed up factors are taken with respect to the values highlighted in bold font.

	Index	DBLP	Geonames	DBpedia	Freebase
		bits/triple	bits/triple	bits/triple	bits/triple
	3T	75.24 (+31%)	71.59 (+32%)	80.64 (+33%)	74.20 (+30%)
	CC	63.54 (+18%)	67.04 (+27%)	66.91 (+19%)	70.46 (+26%)
	2To	56.46 (+8%)	53.23 (+8%)	57.51 (+6%)	55.72 (+6%)
	2Tp	**51.99**	**48.98**	**54.14**	**52.17**
		ns/triple	ns/triple	ns/triple	ns/triple
SPO	*all*	203	221	353	521
SP?	*all*	197	347	11	3
S??	*all*	28	40	10	3
???	*all*	11	13	9	9
S?O	3T,CC	2490 (5.6×)	3767 (7.7×)	1833 (2.6×)	6547 (1.8×)
	2To,2Tp	**445**	**490**	**692**	**3736**
?PO	3T,2To,2Tp	**5**	**5**	**5**	**5**
	CC	12 (2.4×)	15 (3.0×)	16 (3.2×)	14 (2.8×)
??O	3T,CC	12 (2.4×)	12 (2.4×)	12 (2.4×)	10 (2.0×)
	2To	**5**	**5**	**5**	**5**
	2Tp	5 (1.0×)	5 (1.0×)	6 (1.2×)	10 (2.0×)
?P?	3T,2Tp	**9**	**8**	**6**	**6**
	CC	21 (2.3×)	36 (4.5×)	30 (5.0×)	29 (4.8×)
	2To	81 (9.0×)	138 (17.2×)	22 (3.7×)	18 (3.0×)

Figure 3: Comparison between the performance of 3T, CC and 2T indexes, expressed as the total space in bits/triple and in average ns/triple for all the different selection patterns.

The permuted trie index. By looking at the results reported in Figure 3, we can conclude that:
(1) the 3T index is the one delivering best worst-case performance guarantee for all triple selection patterns;
(2) the 2T variants reduce its space of representation by 25 ÷ 33% without affecting or even improving the retrieval efficiency on most triple selection patterns (only one out of the eight possible has a lower query throughput in the worst-case) the cross- compression technique is outperformed by the 2T index layouts for space usage but offers a better worst-case performance guarantee than 2Tp for the pattern ?P?. Therefore, as a reasonable trade-off between space and time, we elect 2Tp as the solution to compare against the state-of-the-art alternatives.

	Index	DBLP	Geonames	DBpedia	Freebase
		bits/triple	bits/triple	bits/triple	bits/triple
	2Tp	**51.99**	**48.98**	**54.14**	**52.17**
	HDT-FoQ	76.89 (+32%)	88.73 (+45%)	76.66 (+29%)	83.11 (+37%)
	TripleBit	125.10 (+58%)	120.03 (+59%)	130.07 (+58%)	—
		ns/triple	ns/triple	ns/triple	ns/triple
	2Tp	5	5	5	5
?PO	HDT-FoQ	12 (2.4×)	13 (2.6×)	14 (2.8×)	13 (2.6×)
	TripleBit	15 (3.0×)	13 (2.6×)	14 (2.8×)	—
	2Tp	**445**	**490**	**692**	**3736**
S?O	HDT-FoQ	1789 (4.0×)	2097 (4.3×)	3010 (4.3×)	0.7×10^7 (2057×)
	TripleBit	11872 (26.7×)	13008 (26.5×)	18023 (26.0×)	—
	2Tp	**197**	**347**	**11**	**3**
SP?	HDT-FoQ	640 (3.2×)	897 (2.6×)	30 (2.7×)	9 (3.0×)
	TripleBit	1222 (6.2×)	927 (2.7×)	42 (3.8×)	—
	2Tp	**28**	**40**	**10**	**3**
S??	HDT-FoQ	110 (3.9×)	154 (3.9×)	29 (2.9×)	9 (3.0×)
	TripleBit	2275 (81.2×)	3261 (81.5×)	490 (49.0×)	—
	2Tp	**9**	**8**	**6**	**4**
?P?	HDT-FoQ	108 (12.0×)	173 (21.6×)	32 (5.3×)	41 (6.8×)
	TripleBit	28 (3.1×)	28 (3.5×)	40 (6.7×)	—
	2Tp	**5**	**5**	**6**	**10**
??O	HDT-FoQ	17 (3.4×)	17 (3.4×)	18 (3.0×)	18 (1.8×)
	TripleBit	24 (4.8×)	60 (12.0×)	24 (4.0×)	—

Figure 4: Comparison between the performance of different indexes, expressed as the total space in bits/triple and in average ns/triple.

Overall comparison. We now compare the performance of our selected solution 2Tp against the competitive approaches HDT-FoQ (Martínez-Prieto et al. 2012) and TripleBit (Yuan et al. 2013). We use the C++ libraries as provided by the corresponding authors and available at https://github.com/rdfhdt/hdt-cpp and https://github.com/nitingupta910/TripleBit, respectively.

Figure 4 reports the space of the indexes and the timings for the different selection patterns, but excluding the ones for SPO and ???: our approach is anyway faster for both by at least a factor of 3× (TripleBit does not support the query pattern SPO).

Concerning the space, we see that the 2Tp index is significantly more compact, specifically by 30% and almost 60% compared to HDT-FoQ and TripleBit respectively, on average across all different datasets (TripleBit fails in building the index on Freebase).

Concerning the speed of triple selection patterns, most factors of improved efficiency range in the interval 2 ÷ 5× and, depending on the pattern examined, we report peaks of 26×, 49×, 81× or even 2057×.

3.3. TIME-SERIES INDEXING

In this section we present a preliminary experimental study we conducted on the problem of *compressing time-series*.

Specifically, a time series is a list of observations recorded in order of time. Formally, a univariate time series is a collection of key-value pairs <T(n), V(n)>, where the key T(n) denotes the exact time in which the observation n was registered whereas the value V(n) is the measurement of a single time-dependent variable. The value V(n) is a floating-point number. A multivariate time series has more than one time-dependent variable, hence, for m variables, it can be expressed in the form:

<T(n), [V_1(n), V_2(n),…, V_m(n)]>.

Time series are widely used in all domains of applied science and engineering that consider temporal measurements. For instance, they can keep track of annual ozone levels, describe the daily blood pressure of a heart patient, or simply monitor the monthly stock market trend. The goal of collecting such data is mainly to extract meaningful information, as well as make forecasts based on previously observed values. Some tasks using exhaustively those data are: rule discovery, classification, clustering, outliers detection and many others in machine learning.

3.3.1. Our proposal: the XOR-Cache algorithm

One of the most relevant characteristics of time-series data is that of *data locality*. More precisely, it is very likely that a time-series contains many repeated values. This happens, for example, because a certain observation V(n) does not change much between close time instants.
Therefore, inspired by the XOR mechanism used by Gorilla, we developed a different approach that takes advantage of such locality in the data.

The high-level idea is to perform compression/decompression using a *window*. Let us consider the compression algorithm. (The decompression algorithm just reverts this procedure.) The window has a fixed size, say 2^k for some $k > 0$, and holds the *last* 2^k compressed items. When compressing a value x, it is first searched in the window: if found, the algorithm just outputs its position in the window using k bits; otherwise it applies the XOR algorithm between x and the value in the window that is more similar to x, with similarity assessed in terms of number of leading and trailing zero bytes. (Higher similarity means higher number of zero bytes.)

Lastly, if the number of leading and trailing zero bytes is not sufficiently large (e.g., it is not larger than 1), the value x is considered to be an *exception* and compressed using an escape mechanism. Lastly, in this preliminary version of the work, we represent the time instants T(n) using the same algorithm Gorilla uses, i.e., delta-encoding, and we leave the exploration of other strategies to represent the time stamps as future work.

Table 1: Real-world Time-Series datasets statistics.

Name	Field	Rows	Features	Unique values %
AMPds2: The Almanac of Minutely Power dataset	Electricity consumption	14,629,292	11	0.002
Bar Crawl: Detecting Heavy Drinking Data Set	Human Activity Recognition	14,057,564	4	16.595
Max-Planck Weather Stations (2009-2018)	Soil Temperature	473,353	32	0.541
Microsoft-Cambridge Kinect Gesture Data Set	Gesture	733,432	80	41.073
Oxford-Man Realized Volatility Indices	Finance	143,397	19	84.290
PAMAP	Physical Activities	3,127,602	44	0.382
UCI Gas sensor array under dynamic gas mixtures	Chemical sensors	2,841,954	18	0.625

Table 2: Decompression speed in Millions of values decompressed per second. Best values are highlighted in bold.

Dataset	Gorilla	Xor-Cache
AMPds2: The Almanac of Minutely Power dataset	**62.39**	133.52
Bar Crawl: Detecting Heavy Drinking Data Set	44.01	**71.62**
Max-Planck Weather Stations (2009-2018)	80.79	**108.81**
Microsoft-Cambridge Kinect Gesture Data Set	62.82	**65.91**
Oxford-Man Realized Volatility Indices	54.57	**66.75**
PAMAP	48.96	**105.22**
UCI Gas sensor array under dynamic gas mixtures	55.48	**75.52**

Table 3: Compression ratios. Best values are highlighted in bold.

Dataset	Gorilla	Xor-Cache
AMPds2: The Almanac of Minutely Power dataset	2.034x	**6.391x**
Bar Crawl: Detecting Heavy Drinking Data Set	1.443x	**2.358x**
Max-Planck Weather Stations (2009-2018)	2.968x	**4.837x**
Microsoft-Cambridge Kinect Gesture Data Set	**1.408x**	1.367x
Oxford-Man Realized Volatility Indices	1.275x	**1.299x**
PAMAP	1.380x	**4.848x**
UCI Gas sensor array under dynamic gas mixtures	1.228x	**3.499x**

3.3.2. Experiments

We conducted a preliminary experimental analysis on very large and publicly available datasets concerning different fields of science, whose statistics are summarized in Table 1.

The experiments were run on a MacBook Pro with 2.8 GHz Quad-Core i7, 6MB of L3 cache and 16GB of RAM running MacOS 10.15.4. The compiler involved was the Apple clang version 11.0.3. Moreover, all the source files have been compiled using the –O3 optimization flag.

For this proof-of-concept experiment, we used a window of size 128, so that every quantity outputs and read by the XOR-Cache algorithm is written in a suitable number of bytes.

In Table 2 and 3 we report, respectively, the decompression speed in millions of values decompressed per second and compression ratios achieved by the XOR-Cache algorithm compared to the strong baseline — Gorilla — that we described in Section 2.3.

The net result of the experiment is that using a window grants both fast decompression speed and better compression ratios than the tested baselines. This is a direct consequent of the highly repetitive nature of the time-series datasets, as already motivated.

In particular, compression is good as most values are found in the window (thus, 1 byte suffices to store the position of a value in the window); decompression speed is very fast as most values are decompressed using just a lookup in the window — an extremely fast operation, with low constant factors, considering that a window of decompressed values fits well in the processor cache. Also, avoiding expensive bit-level manipulations (as Gorilla does), further reduces the processing time.

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 the first level. A graph database does not use 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 a RDF triplestore compliant with the core semantic web W3C specifications (RDF, RDFS, OWL). GraphDB allows users to link diverse data, index it for semantic search and enrich it via text analysis to build big knowledge graphs. GraphDB provides functionalities for all critical seating graph operations (storing, indexing, reasoning, querying, efficient retraction of inferred statements upon update, etc.). The query language used is the implementation of the SPARQL 1.1 specifications plus GeoSPARQL, while connectors with Elasticsearch and Lucence are incorporated in the system.

AllegroGraph[6] 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[7] 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 supports 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] https://franz.com/agraph/allegrograph/

[7] 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[8] 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[9] 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[10] 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 the 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 to have 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[11] and Elasticsearch[12]. 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 [13] 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[14]. 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 faster development of real-time applications.

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

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

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

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

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

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

[14] 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 time stamped 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 period focused on:

1. the design of time and space efficient data structures for indexing text and RDF data that can support a broad range of analytical queries over arbitrary data dimensions;
2. the design of a fast compression algorithm for time-series.

This deliverable presents the software components implementing two novel solutions to index and search massive text and RDF datasets (as per point 1 above). Point 2 is still under development and the deliverable reports on the preliminary design and experiments.

In particular, the solution adopted for inverted indexes consists in a compression technique 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 has been published in a scientific paper by IEEE Transactions on Knowledge and Data Engineering (Pibiri & Venturini 2019b).

The second solution, developed for indexing massive RDF datasets, is based on the well-known trie data structure. In particular, we designed trie-shaped indexes that significantly outperform existing proposals at the-state-of-the-art. The corresponding scientific paper (Perego, Pibiri and Venturini, 2020) has been again published in IEEE Transactions on Knowledge and Data Engineering.

The third solution for time series compression and indexing is still preliminary. However, the results of the experiments conducted so far are promising and, as in the previous cases, we hope to contribute the project partners and the scientific community with a useful and impactful solutions published as open source code and detailed in a publication on a top-tier journal.

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 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. E. & Venturini, R., 2019. Techniques for Inverted Index Compression. CoRR, http://arxiv.org/abs/1908.10598, 35 pages.

Pibiri, G. E. & Venturini, R., 2019. On Optimally Partitioning Variable- Byte Codes. IEEE Transactions on Knowledge and Data Engineering (TKDE), 12 pages.

Perego R., Pibiri, G. E. & Venturini, R., 2020. Compressed Indexes for Fast Search of Semantic Data. IEEE Transactions on Knowledge and Data Engineering (TKDE), 12 pages.

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.

M. A. Martínez-Prieto, M. A. Gallego, and J. D. Fernández, 2012. Exchange and consumption of huge rdf data. In Extended Semantic Web Conference, pages 437–452. Springer, 2012.

P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, 2013. Triplebit: a fast and compact system for large scale rdf data. PVLDB, 6(7):517–528, 2013.

Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. 2010. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In Proceedings of the 19th international conference on World wide web. ACM, 41–50.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. 841–850.

Sandra Álvarez-García, Nieves Brisaboa, Javier D Fernández, Miguel A Martínez- Prieto, and Gonzalo Navarro. 2015. Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems 44, 2 (2015), 439-474.

Nieves Brisaboa, Ana Cerdeira-Pena, Antonio Farina, and Gonzalo Navarro. 2015. A compact RDF store using suffix arrays. In International Symposium on String Processing and Information Retrieval. Springer, 103–115.

Nieves Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k2-trees for com- pact web graph representation. In International Symposium on String Processing and Information Retrieval. Springer, 18–30.

Antonio Fariña, Nieves Brisaboa, Gonzalo Navarro, Francisco Claude, Ángeles Places, and Eduardo Rodríguez. 2012. Word-based self-indexes for natural language text. ACM Transactions on Information Systems (TOIS) 30, 1 (2012), 1.

Kunihiko Sadakane. 2003. New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 2 (2003), 294–313.

Ratanaworabhan, Paruj, Jian Ke, and Martin Burtscher. "Fast lossless compression of scientific floating-point data." Data Compression Conference (DCC'06). IEEE, 2006.

Lindstrom, Peter, and Martin Isenburg. "Fast and efficient compression of floating-point data." IEEE transactions on visualization and computer graphics 12.5 (2006): 1245-1250.

Pelkonen, Tuomas, et al. "Gorilla: A fast, scalable, in-memory time series database." Proceedings of the VLDB Endowment 8.12 (2015): 1816-1827.

7. APPENDIX

7.1. SOFTWARE FOR INVERTED INDEXES

The code is available at https://github.com/jermp/opt_vbyte.

7.1.1. Compiling 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.

7.1.2. 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.

7.1.3. From RDF to the input data format

We can use the script rdf_to_index.sh to convert an 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.

7.1.4. 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.

1.1.1.1.1 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.

1.1.1.1.2 Example 2.

The command

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

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

1.1.1.1.3 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. SOFTWARE FOR TRIE-BASED INDEXES

The code is available at https://github.com/jermp/rdf_indexes.

7.2.1. Compiling the code

The code is tested on Linux with gcc 7.3.0 and on Mac 10.14 with clang 10.0.0. To build the code, CMake and Boost are required.

The code has few external dependencies (for testing, serialization and memory-mapping facilities), so clone the repository with

```
> git clone --recursive https://github.com/jermp/rdf_indexes.git
```

If you have cloned the repository without --recursive, you will need to perform the following commands before compiling:

```
> git submodule init
> git submodule update
```

To compile the code for a release environment (see file CMakeLists.txt for the used compilation flags), it is sufficient to do the following:

```
> mkdir build
> cd build
> cmake ..
> make
```

Hint: Use make -j4 to compile the library in parallel using, e.g., 4 jobs.

For a testing environment, use the following instead:

```
> mkdir debug_build
> cd debug_build
> cmake .. -DCMAKE_BUILD_TYPE=Debug -DUSE_SANITIZERS=On
> make
```

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.

7.2.2. Input data format

The library works exclusively with integer triples, thus the data has to be prepared accordingly prior to indexing and querying. We assume the RDF triples have been mapped to integer identifiers and sorted in different permutations of the subject (S), predicate (P) and object (O) components.
To build an index we need the following permutations: SPO, POS and OSP. Also the permutation OPS may be needed to build some types of indexes. Each permutation is represented by a separate file in plain format, having an integer triple per line with integers separated by whitespaces. As an example, if our dataset is named RDF_dataset, we need the following files:

- RDF_dataset.spo
- RDF_dataset.pos
- RDF_dataset.osp
- RDF_dataset.ops

We also need a metadata file containing some useful statistics about the data. This file must be named RDF_dataset.stats. Also this file is in plain format and contains 7 integers, one per line:

1. total number of triples
2. distinct subjects

3. distinct predicates
4. distinct objects
5. distinct S-P pairs
6. distinct P-O pairs
7. distinct O-S pairs

The next section details how this data format can be created automatically from a given RDF dataset in standard N-Triples (.nt) format. (See also https://www.w3.org/TR/n-triples.)

7.2.3. Preparing the data for indexing

The folder scripts contains all the python scripts needed to prepare the datasets for indexing.
Assume we have an RDF dataset in standard N-Triples format, additionally compressed via gzip. For the following example, assume to work with the dataset provided in the folder test_data: wordnet31.nt.gz. (This dataset has been downloaded from http://www.rdfhdt.org/datasets and extracted using the HDT (Martínez-Prieto et al., 2012) software at http://www.rdfhdt.org/downloads.)

To prepare the data, it is sufficient to follow the following steps from within the scripts folder.

1. Extract the vocabularies.

```
> python extract_vocabs.py ../test_data/wordnet31.nt.gz -S -P -O
```

This script will produce the following files: wordnet31.subjects_vocab, wordnet31.predicates_vocab and wordnet31.objects_vocab.

3. Map the URIs to integer triples.

```
> python map_dataset.py ../test_data/wordnet31.nt.gz
```

This script will map the dataset to integer triples, producing the file wordnet31.mapped.unsorted.

4. Sort the file wordnet31.mapped.unsorted materializing the needed permutations.

```
> python sort.py ../test_data/wordnet31.mapped.unsorted wordnet31
```

This script will produce the four permutations, one per file: wordnet31.mapped.sorted.spo, wordnet31.mapped.sorted.pos, wordnet31.mapped.sorted.osp and wordnet31.mapped.sorted.ops.

5. Build the file with the statistics.

```
> python build_stats.py wordnet31.mapped.sorted
```

This script will create the file wordnet31.mapped.sorted.stats.

7.2.4. Building an index

With all the data prepared for indexing as explained before, building an index is as easy as:

```
> ./build <type> <collection_basename> [-o output_filename]
```

For example, the command:

> ./build pef_3t ../test_data/wordnet31.mapped.sorted -o wordnet31.pef_3t.bin

will build a 3T index, compressed with partitioned Elias-Fano (PEF), that is serialized to the binary file wordnet31.pef_3t.bin.

See also the file include/types.hpp for all other index types. At the moment we support the following types: compact_3t ef_3t vb_3t pef_3t pef_r_3t pef_2to pef_2tp

7.2.5. Querying an index

A triple selection pattern is just an ordinary integer triple with k wildcard symbols, for $0 \leq k \leq 3$. In the library, a wildcard is represented by the integer -1. For example, the query pattern

13 549 -1

asks for all triples where subject = 13 and predicate = 549. Similarly

-1 -1 286

asks for all triples having object = 286.

If you do not have a query log with some triple selection patterns of this form, just sample randomly the input data with (use gshuf instead of shuf on Mac OSX)

> shuf -n 5000 ../../test_data/wordnet31.mapped.unsorted > ../../test_data/wordnet31.mapped.unsorted.queries.5000

that will create a query log with 5000 triples selected at random.

Then, the executable ./queries can be used to query an index, specifying a querylog, the number and position of the wildcards:

> ./queries <type> <perm> <index_filename> [-q <query_filename> -n <num_queries> -w <num_wildcards>]

The arguments <perm> and -w <num_wildcards> are used to specify the triple selection patterns. <perm> is an integer 1..3 indicating the S-P-O permutation where <num_wildcards> symbols are set to wildcards (starting from the right). We use the convention that perm = 1 specifies SPO, perm = 2 specifies POS and perm = 3 specifies OSP.

Therefore, we have:

- perm = 1 and -w 0 <=> SPO
- perm = 1 and -w 1 <=> SP?
- perm = 1 and -w 2 <=> S??
- perm = 2 and -w 1 <=> ?PO
- perm = 2 and -w 2 <=> ?P?
- perm = 3 and -w 1 <=> S?O
- perm = 3 and -w 2 <=> ??O
- any perm and -w 3 <=> ???

For example

> ./queries pef_3t 1 wordnet31.pef_3t.bin -q ../test_data/wordnet31.mapped.unsorted.queries.5000 -n 5000 -w 1

will execute 5000 SP? queries.

7.2.6. Statistics

The executable ./statistics will print some useful statistics about the nodes of the tries and their space occupancy:

> ./statistics <type> <index_filename>

For example

> ./statistics pef_2tp wordnet31.pef_2tp.bin

7.2.7. Testing

Run the script test/check_everything.py from within the ./build directory to execute an exhaustive testing of every type of index.

> python ../test/check_everything.py ../test_data/wordnet31.mapped.sorted . wordnet

This script will check every triple selection pattern for all the different types of indexes.

NOTE: See also the directory ./test for further testing executables.

7.3. SOFTWARE FOR TIME-SERIES

The code is available at https://github.com/andybbruno/LZ-XOR.

7.3.1. Compiling the code

The code has been tested on MacOS 10.15.4 using clang version 11.0.3.

No dependencies are needed. Just clone the repository and execute:

> make all

7.3.2. Input data format

The algorithm can process any .csv file containing numbers only.
You need first to convert the .csv into a .bin file using the csv_to_bin utility as follows:

> cd util
> ./csv_to_bin.o path/to/MY_DATASET.csv

Please note: the first column will be interpreted as the timestamp, the rest will be interpreted as values.

7.3.3. Compression

To run a compression test of a `.bin` file, execute the following commands:

```
> cd test

> ./compression.o path/to/MY_DATASET.bin
```

This will produce a file called ``compressed_data.lzx``

7.3.4. Decompression

To decompress the file ``compressed_data.lzx``, run the command:

```
> ./decompression.o
```

7.4. 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/