Compressing and Maintaining Statistics Information about Resource Occurrences in a Distributed RDF Store

Daniel Janke[1] and Steffen Staab[1,2]

[1] Institute for Web Science and Technologies
Universität Koblenz-Landau, Germany
{danijank,staab}@uni-koblenz.de
http://west.uni-koblenz.de/
[2] Web and Internet Science Group
University of Southampton, UK
s.r.staab@soton.ac.uk
http://wais.ecs.soton.ac.uk/

Abstract. In distributed RDF stores triples are assigned to one or several storage and compute nodes. In order to perform query planning and optimization, statistical information about the occurrences of IRIs and literals on the individual storage and compute nodes is needed. In this paper, we present our novel compressed storage format for statistical information that can be updated with a single read and write operation if resources occur on few storage and compute nodes only. In our experiments this novel storage format reduced the time to collect statistical information by up to 97% and the required space by up to 99%.

1 Introduction

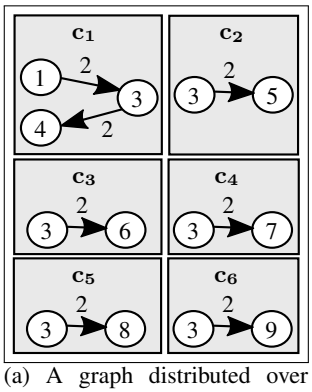
Organizations with needs for massive storage and management of RDF data, e.g. BBC or Wikidata, operate distributed RDF stores [3]. Among the targeted benefits of distributed RDF data management solutions are better fault tolerance and faster querying.

The data loading in distributed RDF stores typically includes: (i) data encoding: each IRI and literal is replaced by a unique numerical identifier (called resource ID), (ii) data distribution: all triples are assigned to compute nodes[3], (iii) global data indexing: this index keeps track how frequently resources occur on the individual compute nodes and (iv) local data indexing: compute nodes index their local data. An exemplary encoded RDF graph that is distributed over compute nodes c_1 to c_6 is shown in Fig. 1a.

One challenge of distributed RDF stores is the distributed query processing. Thereby, the query planner has to decide which subqueries can be executed locally and in which order the results should be joined [4]. To make this decision the query planner needs a global index that stores how frequently the resources occur on the individual nodes. Ideally, an index provides statistics about frequencies and interdependencies between resources, such as Hexastore [5] provides six index structures to allow for immediately accessing data corresponding to any SPARQL triple pattern. If data size is so large that it requires distributed storage and processing, such strategies may become infeasible, e.g., the indices of Hexastore are four times larger than the original dataset. In fact, even indices that only count and point to individual resources may grow so large and unwieldy that the advantages of distributed processing may be jeopardized. Therefore, we have developed a global index that can be compressed so much that it would fit into main memory of a central master compute node.

Fig. 1 depicts our running example that illustrates the problem and sketches our solution. Fig. 1b shows the resource frequency table. Columns 0-5, 6-11 and 12-17 store the information how frequently resources occur as subject, property or object on the six

[3] For a better comprehensibility, we omit the differentiation between storage and compute nodes.



(a) A graph distributed over compute nodes c_1 to c_6.

Resource	Subject						Property						Object					
	c_1	c_2	c_3	c_4	c_5	c_6	c_1	c_2	c_3	c_4	c_5	c_6	c_1	c_2	c_3	c_4	c_5	c_6
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	2	1	1	1	1	1	0	0	0	0	0	0
3	1	1	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

(b) The resource frequency table of the example from Fig. 1a.

Fig. 1: A distributed example graph and its resource frequency table.

compute nodes, respectively. Each row represents the occurrences of a single resource as subject, property or object on the individual compute nodes. For instance, the resource 1 occurs only one time as subject on compute node c_1. Therefore, the resource frequencies row 1 has a 1 in the first column and all other columns are 0.

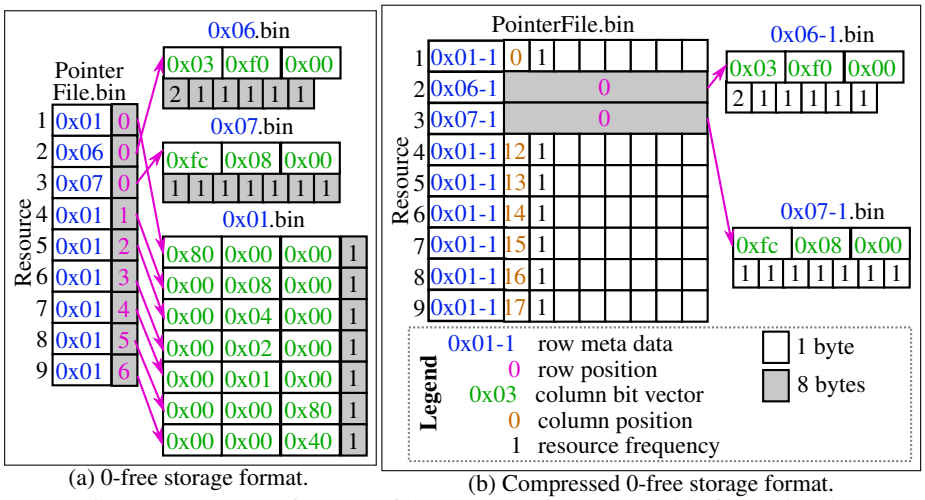
The resource frequency table created during our experiments described in Sec. 3 stored in 90% of all cells 0s. Furthermore, 99.97% of all remaining resource frequencies are smaller than 255 and thus, can be encoded with only 1 byte (see Sec. 3). Thus, the resource frequency table has a high potential to be compressed efficiently. Nevertheless, compression techniques like run-length encoding have the disadvantage that in the presence of updates large portions of the resource frequency table needs to be decompressed, updated and compressed again afterwards. Therefore, updates are expensive.

Our contribution is a novel storage format that reduces the disk space to store the resource frequency table while being updateable by single read and write operations for most resources (see Sec. 2). This is achieved by (a) not storing 0 frequencies, (b) allowing for integers with variable length and (c) by bundling compressed rows of the same length. In our experiments (see Sec.3), our novel storage format could reduce the time to collect the resource frequencies by up to 97% and the required disk space by up to 99%. An open source implementation can be found at [1].

2 Compressed 0-free Storage Format

0-free Storage Format. To avoid storing resource frequencies of 0, we use one bit vector per resource (called column bit vector) that indicates for each column whether a value >0 is present, followed by the corresponding values. In our running example there exist 18 columns. Thus, we need a bitmap of 3 bytes. Using a hexadecimal notation for the bitmap, the compressed resource frequencies row of resource 1 from Fig. 1b is $[0x80, 0x00, 0x00, 1]$.

Maintaining Constant Access Time. If the resource IDs form a consecutive sequence, the uncompressed resource frequency table has the advantage that by knowing the total number of compute nodes and the resource IDs the position of the corresponding resource frequencies rows can be computed. By not storing 0s, the resource frequencies rows of the resources can have different lengths. Therefore, pointers to the position of the resource frequencies row of each row are required. We store the pointers in a separate file to allow for adding new entries at its end.



(a) 0-free storage format.

(b) Compressed 0-free storage format.

Fig. 2: Two storage formats of the resource frequency table from Fig. 1b.

Reduction of Fragmentation. Storing all compressed resource frequencies rows in a single file has the disadvantage that if a row frequency entry is increased from 0 to 1 or more, the corresponding resource frequencies row needs to be appended to the end of the file. As a consequence, the previously occupied bytes become unused leading to a fragmentation of the file. To simplify the reusage of freed bytes, we split the file into several files based on the number of columns with a value >0. This leads to files, in which each resource frequencies row has the same length. Due to this property, a single bit vector for every file is enough to keep track of free and occupied resource frequencies rows. This bit vector can be efficiently compressed using run-length encoding which allows to keep them in main memory.

To keep track of how long the resource frequencies rows in a file are, the number of columns that have values >0 needs to be stored. In our running example a single byte (called row meta data) can store this number. Instead of storing the row meta data for each file, the files are named by its hexadecimal string. To locate the resource frequencies row of a resource, the pointer file entries are extended by the row meta data. Fig. 2a shows the table from Fig. 1b compressed with the presented techniques.

Compressed 0-free Storage Format. We apply the following compressions:

Variable-bytes Integers. To reduce the number of bytes required for the resource frequencies we allow for integers of 1 to 8 bytes. To maintain the property that all resource frequencies rows within a file have the same length, we split up each file into 1-8 files depending on the minimal number of bytes that are required to store all resource frequencies within a resource frequencies row. To indicate how many numbers of bytes are required, the row meta data is extended by 3 bits. In our running example, the row meta data consists of 5 bits for the number of used columns and 3 bits for the integer length.

Reducing Size of Column Bit Vector. In our example, the column bit vector requires 3 bytes, whereas the position of a specific column requires only 1 byte. Therefore, all resource frequencies rows that have ≤ 3 columns with a value >0 list the indices of the used columns instead of using column bit vectors. E.g., only the first column of resource 1 has a value >0. Therefore, the resource frequencies row is stored as $[0, 1]$ with 0 as the column position. Since this compression strategy applies to all resource frequencies rows within one file, no additional row meta data is required.

Integrating Resource Frequencies Rows into Pointer File. In the case of resource 1 the storage of the resource frequencies row consumes only two bytes whereas the row position in the pointer file consumes 8 bytes. To reduce the storage size and the number of disk operations, we can store the resource frequencies row directly in the space reserved for the row position. With the help of the row meta data and the number of compute nodes it can be computed whether a row position or the actual row data is stored in the pointer file. The resulting compressed storage format is shown in Fig.2b. In order to be more comprehensive, the row meta data is split into the number of used columns and the integer length separated by a -.

Runtime Complexity. Since the disk I/O is the most time consuming operation, we discuss the runtime complexity by investigating the number of I/O operations. To read the resource frequencies row of a resource, in the worst case a lookup in the pointer file and a lookup in one of the resource frequencies row files is required. The worst case for a frequency update appears, if in our running example the frequency of resource 2 in the first column would be incremented. In this case the initial read of the old value consumes two read operations. During the update of the resource frequencies row, its size is increased by one byte. Therefore, it needs to be moved from file 0x06-1.bin to 0x07-1.bin. Also, its entry in the pointer file needs to be updated. Thus, two read and two write operations are required. As our experiments indicate, the resource frequencies rows of most resources can be integrated in the pointer file. In these cases reading consumes a single read operation and incrementing one read and one write operations.

3 Evaluation

For our evaluation we use the first 100M triples of the billion triple challenge 2014 dataset [2]. These triples are assigned to 40 compute nodes based on the hash of their subjects. When the assignment is done, we iterate over all triples and increment the frequencies of the occurring resources in the resource frequency table. As a baseline approach we store the uncompressed resource frequency table in a single file using 8 bytes for each cell. All experiments are performed on a compute node with 1 CPU core, 2 GB RAM and 300 GB disk space.

The baseline approach required more than 22 hours to count the frequencies of all resources in the partitioned dataset. In contrast to this, our novel compression format required only 42 minutes due to its compression format that supports updates efficiently. This is a reduction of approximately 97%. In terms of storage consumption, the baseline approach required more than 36 GB, whereas our novel compression format required only 397 MB in total. This is a reduction by 99% and would allow for storing it in main memory. We only evaluated insertions of new resources and incrementations of resource frequencies. Deleting resources and decrementing their frequencies as well as an in-memory implementation of our novel approach will be evaluated in future work.

References

1. Koral. `https://github.com/Institute-Web-Science-and-Technologies/koral`, accessed: 2018-05-29
2. Käfer, T., Harth, A.: Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2014/ (2014)
3. Kiryakov, A., et al.: The Features of BigOWLIM that Enabled the BBC's World Cup Website. In: Workshop on Semantic Data Management (SemData@VLDB 2010) (Sep 2010)
4. Sakr, S., Wylot, M., Mutharaju, R., Le Phuoc, D., Fundulaki, I.: Linked Data: Storing, Querying, and Reasoning. Springer International Publishing, Cham, 1 edn. (2018)
5. Weiss, C., et al.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1), 1008–1019 (2008), `http://www.vldb.org/pvldb/1/1453965.pdf`