Time And Relative Dimensions In Semantics
OWL – Bigger On The Inside?

Simon E. Spero

Criticollab LLC
ses@criticollab.com *

Abstract. The OWL 2 recommendations define a number of formats for storing and exchanging ontologies. OWLAPI is a widely used Java framework for working with such ontologies. Relatively little time has been spent on optimizing for space.
This paper examines size optimizations for the OWLAPI that have been implemented in current releases of the OWLAPI. This paper also looks at the benefits of generating sorted output for version control and compression, and examines the effects on size of different approaches to packaging multiple ontologies.

1 Introduction

The work described in this paper began as puzzle: why does a half gigabyte owl document use over two gigabytes of heap space when loaded by the OWLAPI? Why does the document use half a gigabyte in the first place? Is OWL really bigger on the inside?

1.1 Java Internals

Assumptions In this paper, all calculations and measurements of Java memory usage are based on a 64-bit Hotspot Java Virtual Machine, with a heap size of less than 32GB, and compressed ordinary object pointers (oops) enabled. Compressed oops [12] are an optimization that allow for object references to be stored in 32-bits, and are enabled by default for heap sizes $<$ 32GB. Similar approaches are used in other 64-bit VM implementations [2,10].

All Java Objects must be aligned on an 8-byte boundary. Native data types must be aligned at the appropriate boundaries (e.g. `int`s must start on a 4 byte boundary, `double`s an 8 byte-boundary, etc.). If necessary, padding will be added to ensure appropriate alignment.

Java Object layout. The layout of a plain Java object layout is shown in figure 1. All objects have a fixed overhead of 12 bytes, and the minimum size of an object is 16 bytes. The 4 bytes immediately following the header fields are available for use.

* I would like to thank Chris Mungall, Matthew Horridge, Ignazio Palmisano, and Peter Ansell for their support and feedback during the course of this regeneration.

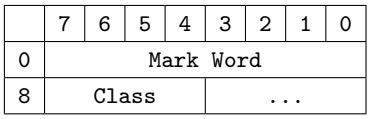

Fig. 1. Java hotspot object layouts

The layout of a Java array is shown in figure 2. The layout is the same as for a plain object, with the addition of a 4 byte integer holding the array length. The contents of the array follow, with the overall array size padded to an 8-byte boundary. Thus, the size of an empty array is 16 bytes. The minimum size of a non-empty array is 24 bytes.

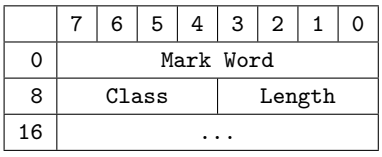

Fig. 2. Hotspot array layout

The following example illustrates how the single character `String`, `"a"` is stored in memory.

The `String` object in Fig. 3 contains two fields: `hash`, an `int`, and `value`, a 4 byte reference to an array of `char`. With 12 bytes of header, 8 bytes of payload, and 4 bytes of padding, the `String` object directly consumes a total of 24 bytes of memory.

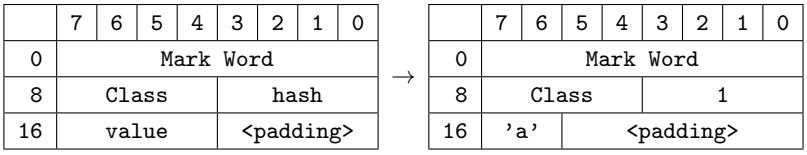

Fig. 3. `String` and `char` array

The value field in the string object points to a character array of length 1. This array has a 16 byte header, a 2-byte payload, and six bytes of padding, consuming another 24 bytes of memory.

Thus, the single character string "a" uses 48 bytes of heap space. The same amount of memory would used for strings up to four characters in length. Future versions of java may use byte arrays to store string values that only characters

from ISO-8859-1. With this change, "a" will still use 48 bytes of heap space; however, we will be able to store strings up to 8 characters long in the same space.

2 The Inside

2.1 Where Has All the Memory Gone?

The first step in reducing the amount of memory used by OWLAPI was to find out how the memory was being used. This task was made much easier by the use of the Eclipse Memory Analyzer (MAT) [1]. This tool provides a graphical interface for analyzing heap dumps from java processes, allowing one to quickly find out which classes and objects are using the most memory, both directly, and by reference.

Looking at a few large ontologies, primarily the NCBI Taxonomy Ontology (NCBITaxon) [13] and the Gene Ontology [9], quickly revealed three areas in need of optimization; in order of significance, these were internal indexes, literals, and unnecessary fields in axioms and expressions.

2.2 Certain Non-magical Indexes

OWLAPI was using the `MultiMap` implementation from Google's guava [4]library. This package provides a great deal of flexibility in allowing for different features, as such as maintaining linked or unlinked keys and values, and allowing for different implementations of Maps and Sets to be used.

Switch to unlinked MultiMaps. Initial analysis using NCBITaxon (455MB of OWL FSS) revealed that OWLAPI was using fully linked MultiMaps as indices, which preserve insertion order across globally as well as for individual sets, with a correspondingly large overhead (row (a), Table 1). Since there was no real need to preserve insertion order of any kind, switching to unlinked keys, with unlinked Java `HashSet` offered a very simple way to save over 10% of the total ontology size (row (b)).

Use Alternate Implementation of Maps and Sets. Java's default implementation of Hash Maps and Hash Sets are rather space inefficient. Switching to alternative implementation, such as `THashMap` and `THashSet` from the trove4j library [14] can provide substantial reductions in memory use (rows c–f, Table 1).

Trove based indexes using a load factor of 0.75 (row (e)) were implemented for OWLAPI version 4.0.0.

Trove does have some slight speed tradeoffs compared to other alternative collection packages. These performance issues are primarily driven by the use of prime numbers for hash table sizes, rather than powers of two. This choice requires integer division, rather than a simple bit shift - a much more expensive operation. Future work will measure the performance tradeoffs of other high performance collections libraries.

Table 1. Ontology internals size for different MultiMap configurations

	Keys	Values	Size	Scaled Size
(a)	Linked	Linked Set	1,696.0 MB	1.000
(b)	Unlinked	Unlinked HashSet	1,523.6 MB	0.898
(c)	HashMap	THashSet (load=0.85)	954.2 MB	0.563
(d)	THashMap	THashSet (load=0.50)	999.5 MB	0.589
(e)	THashMap	THashSet (load=0.75)	902.3 MB	0.532
(f)	THashMap	THashSet (load=0.85)	901.7 MB	0.532

Optimizing for Small Sets. The fixed overhead of Java arrays and objects is most noticeable when it affects a large number of very small objects. This occurs very frequently in OWLAPI indices.

The initial load of NCBITaxon requires 1,962,357 sets of index postings. Of these, 981,682 (50%) contain only a single entry, and 923,877 (47%) contain only two or three entries.

A `THashSet` with a maximum capacity of 3 elements requires 88 bytes.

The standard Java Library provides a special implementation for single element sets,which 16 bytes per set. A SmallSet class, which can store up to 3 entries in 24 bytes.

The indexing code was changed to switch between implementations as the size increased or decreased. For the NCBITaxon example, the use of singleton sets saves 67.4MB, and of small sets saves 56.4MB, for an additional saving of 123.8 MB.

This change was implemented for OWLAPI version 4.0.1.

2.3 Improvements to OWLAPI Implementation Classes

Datatype Specific Literal Implementations . The generic implementation of OWL Literals uses four fields; a value, a datatype, a language code, and a cached hashcode, each of which uses 4 bytes. Since this is an even number of fields, an extra four bytes of padding is required, giving a total size for each such literal of 32 bytes.

Version 4.0.0 introduced specialized implementations for the most common data types, such as strings, ints, booleans, etc. These literals only require a single field, giving an object size of 16 bytes.

NCBITaxon uses 3,243,649 literals; this change thus saves 49.5MB.

Remove Unnecessary Fields from Axioms. Every OWL Object in version 4.0.0 has two fields for caching the signature, and the set of of anonymous individuals referenced by that object. This includes entities such as OWL Classes and Datatypes, as well as axioms such as annotation assertions where the cost of regenerating the signature is trivial.

All Axioms are have an additional field, which caches a copy of the Negation Normal Form (NNF); something that can usually cached separately by applications that need it, and again, something that is not too hard to regenerate for axiom types like annotation assertions.

For version 4.0.1, the NNF is no longer cached, and OWL Objects that can easily regenerate them do not cache their signature or anonymous individual set. For annotation assertions this reduces the object size from 48 bytes to 32 bytes.

For NCBITaxon, these changes save 62.6MB on annotation assertions alone.

2.4 Strings and Things

As we saw in the introduction to Java object layouts, Java Strings have an extremely high overhead, especially for short strings, and strings using only characters with code points less than 256. This overhead can be reduced by storing direct references to character or byte arrays in Literals, and only generating Strings when necessary. These Literals can implement the CharSequence interface, which can be used instead of String in some situations, for example, when appending to a String Builder, or matching a regular expression.

The overhead of different representations of strings can be seen in table 2. This table gives the total sizes of the strings in NCBITaxon when stored in different forms. NCBITaxon contains 3,243,649 string literals, 1,266,001 of which are unique.

Table 2. Space used by literals in NCBITaxon Ontology

Form	Size	vs UTF-8	Unique Size	vs UTF-8
Raw size (UTF-8)	52,025,624	–	31,889,949	–
Raw size (UTF-16)	104,051,248	2	63,779,898	2
Java `char[]`	166,874,432	3.2	87,622,360	2.7
Java `byte[]`	112,000,568	2.2	56,539,424	1.8
Java `String`	244,722,008	4.7	118,006,384	3.7
OWLAPI Literal (String)	296,620,392	5.7	138,262,400	4.3
OWLAPI Literal (direct `byte[]`)	163,898,952	3.2	76,795,440	2.4

An experimental version of string-type literals has been implemented which does not use java strings, and which use either a byte array or a char array depending on the characters used in the value.

This implementation has been further extended by creating a specialized implementation for string valued annotation assertions which directly holds a reference to the array. This implementation has not been released, but may be a selectable option for version 4.1.0.

The memory usage for these optimizations provides a useful view of the progress that has been made so far.

Table 3. Evolution of Ontology Internals Sizes

Version	Size
Start	1,696.0 MB
Version 4.0.0	827.7 MB
Version 4.0.1	624.2 MB
Byte array literals	497.7 MB
Byte array assertions	443.1 MB

3 The Outside

3.1 OWLAPI vs. Version Control Systems

It has been noted that OWL serializations are not ideal for use with standard version control systems like subversion or git, and that RDF formats behave particularly badly [3,8]. Many of these problems arise from the fact that, unlike formats such as OBO, OWL does not have a stable ordering for outputting individual axioms [8]. The files produced by the OWLAPI were particularly good examples/horrible warnings of this issue.

Comparing number of insertions and deletions reported by a version control system for the OBO and OWL versions of the Gene Ontology is quite illustrative. Some representative counts are shown in Table 4. The only change made on June 15th was the automatic generation of a new version IRI. The number of changes in the RDF/XML file seems to be *almost* completely random.

Statistical analysis indicates that, the OBO and OWL sizes are significantly correlated ($p < 0.001$, df = 777, R^2=0.18)[1].

If 4 cases where the entire OBO file was changed, and then changed back, are removed, there is no statistically significant correlation (p> 0.35, df= 773, R^2=0.0011).

Table 4. Change counts for Gene Ontology for OBO and original OWL files

Date	**OBO**		**Original OWL**	
	Insertions	Deletions	Insertions	Deletions
14 Jun 2014	28	5	164,305	164,256
15 Jun 2014	1	1	163,941	163,941

The OWLAPI rendering code grouped axioms by entity, and although axioms were collected in unordered hash tables, this does not explain the amount of

[1] Statistics were calculated using the total number of changes (insertions + deletions)

randomness in the output. Sorting every unordered collection before output, so that entities, axioms, and expressions were visited in a predictable order, did not resolve the issue.

The *almost* completely random changes in the RDF/XML output can be partially explained by the method used to allocate identifiers for blank nodes. This turned out to be the Java `System.identityHashCode()` method... which is a pseudo-random number generator. Axioms, and other anonymous individuals, were particularly likely to change position.

Changing the assignment mechanism to generate monotonically increasing identifiers, combined with the predictable order in which identifiers were needed, produced much more satisfactory results, as seen in Table 5.

Table 5. Change counts for Gene Ontology for OBO and ordered OWL files

Release date	**OBO**		**Ordered OWL**	
	Insertions	Deletions	Insertions	Deletions
14 Jun 2014	28	5	54	5
15 Jun 2014	1	1	1	1

The counts are strongly correlated with changes in the OBO file. With the four extreme data points included, ($p < 0.001$, df=777, R^2=0.986).

Excluding these points still produced a highly significant correlation ($p < 0.001$, df=773, $R^2 = 0.6387$)

3.2 Compressing OWL Files

The OWLAPI has long supported the use of GZip and Zip compressed files. However, these formats do not give very good compression ratios when compared to more modern algorithms such as LZMA, as used by packages such as xz [16].

Table 6 shows the file sizes for different serialization formats and compression choices, using the version of the Gene Ontology file used for the example in Section 3.3.

Sizes are given for Binary OWL [5], RDF/XML, Functional Style Syntax (FSS), and lexically sorted FSS, using no compression, zip, gzip and xz.

There are a few unexpected results in this table. The most surprising result is how Binary OWL is by far the smallest uncompressed format, yet gives by far the largest compressed files. The larger compression dictionary size for xz helps a little, but the results are still less than ideal. This phenomena appears to be consistent across a variety of different files, and requires further investigation.

XZ decompression is supported in OWLAPI version 4.0.2 and later.

Table 6. File Sizes for Different Compression and Serialization Formats.

Format	No compression	zip	gzip	xz
Binary OWL	28.0 MB	8.8 MB	8.8 MB	4.4 MB
RDF/XML	111.0 MB	5.7 MB	5.7 MB	3.2 MB
FSS	58.0 MB	5.3 MB	5.3 MB	3.1 MB
FSS (sorted)	58.0 MB	4.8 MB	4.8 MB	2.7 MB

3.3 Bundles Of Ontologies

After receiving requests to add support for OWL/ZIP [7] to OWLAPI, and having an immediate need to support packaging of multiple ontologies for distribution, I have been investigating the size and performance implications of different approaches to bundling ontologies.

The tests below were run using a decomposition of the Gene Ontology [9] OWL release from May 25th 2014[2]. All OWL files were formatted using the ordered output described in Section 3.1.

The ontology contains 486,054 axioms, 90,706 of which were logical. The ontology was decomposed into $\bot$-local atoms using OWLAPI Tools,[15]. This decomposition generated 38,801 atoms.

Table 7 shows the raw and compressed size of the serialized text before and after generating ontology documents for each atom.

Table 7. Unpackaged Sizes

Transformations	Size (RDF)	Size (FSS)
original	111.0M	58.0M
+ xz	3.2M	3.1M
+ sort + xz	–	2.7M
File per atom (du)	644.0M	367.0M
+ xz	153.0M	153.0M
File per atom (cat)	568.0M	295.0M
+ xz	61.0M	47.0M

Compress then Archive, or Archive then Compress? Table 8 clearly shows the difference in compression ratios between compressing files individually and then packaging the compressed files (zip -9, or xz + tar), vs packaging the

[2] http://purl.obolibrary.org/obo/go/releases/2014-05-25/go.owl

Table 8. Packaged Sizes

Transformations	Size (RDF)	Size (FSS)
zip -0	576.0M	302.0M
zip -9	70.0M	53.0M
xz + zip -0	69.0M	55.0M
zip -0 + xz	11.0M	8.5M
tar	596.0M	323.0M
xz + tar	90.0M	74.0M
tar + xz	6.3M	5.8M
Annotations	196.0M	143.0M
+ xz	4.5M	4.5M
+ sort + xz	–	4.0M

uncompressed files, and compressing the complete archive (zip -0 + xz, or tar + xz).

When the files are compressed individually, the compression dictionary is reset for each file. This means that any commonalities across files cannot be exploited. Thus we see an order of magnitude difference for RDF/XML files between zip -9 (70 MB)/xz + zip -0 (69 MB) compared to tar.xz (6.3 MB).

Annotate then Compress? One alternative to using an archive of multiple ontologies to distribute a decomposed ontology is create a fresh ontology with copies of all the axioms in the original, annotated to indicate which atom they belong to. The new ontology would also contain individuals to represent each atom and its dependency structure.

This approach produces the smallest files; the annotated axioms can be split up into multiple ontologies, or tools can reload their atomic decomposition structures directly.

4 Conclusion

If, as de Saint Exupéry put it, perfection is attained not when there is nothing more to add, but when there is nothing more to take away, then the OWLAPI is not yet perfect. It is, however, getting Smaller On The Inside.

References

1. Buchen, A.: Eclipse memory analyzer, http://www.eclipse.org/mat/
2. Cox, J.S., Quirk, A., Inglis, D., Grcevski, N., Agarwal, P.: Introducing WebSphere Compressed Reference Technology. IBM WebSphere Application Server V7 64-bit Performance White Paper, IBM (2008), ftp://public.dhe.ibm.com/software/webserver/appserv/was/WAS_V7_64-bit_performance.pdf

3. Dumontier, M.: diff'ing OWL files with version control systems (April 2014), https://www.w3.org/community/owled/2014/04/02/diffing-owl-files-with-version-control-systems/
4. Google: guava, https://github.com/google/guava
5. Horridge, M., Redmond, T., Tudorache, T., Musen, M.A.: Binary OWL. In: Rodriguez-Muro et al. [11], http://ceur-ws.org/Vol-1080/owled2013_12.pdf
6. Keet, C.M., Tamma, V.A.M. (eds.): Proceedings of the 11th International Workshop on OWL: Experiences and Directions (OWLED 2014) co-located with 13th International Semantic Web Conference on (ISWC 2014), Riva del Garda, Italy, October 17-18, 2014, CEUR Workshop Proceedings, vol. 1265. CEUR-WS.org (2014), http://ceur-ws.org/Vol-1265
7. Matentzoglu, N., Parsia, B.: OWL/ZIP: distributing large and modular ontologies. In: Keet and Tamma [6], pp. 37–48, http://ceur-ws.org/Vol-1265/owled2014_submission_6.pdf
8. Mungall, C.: The perils of managing OWL in a version control system (March 2014), https://douroucouli.wordpress.com/2014/03/30/the-perils-of-managing-owl-in-a-version-control-system/
9. Mungall, C., Dietze, H., Osumi-Sutherland, D.: Use of OWL within the Gene Ontology. In: Keet and Tamma [6], pp. 25–36, http://ceur-ws.org/Vol-1265/owled2014_submission_5.pdf
10. Nilsson, T.: The JRockit Blog - Understanding Compressed References, https://blogs.oracle.com/jrockit/entry/understanding_compressed_refer
11. Rodriguez-Muro, M., Jupp, S., Srinivas, K. (eds.): Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED 2013) co-located with 10th Extended Semantic Web Conference (ESWC 2013), Montpellier, France, May 26-27, 2013, CEUR Workshop Proceedings, vol. 1080. CEUR-WS.org (2013), http://ceur-ws.org/Vol-1080
12. Rose, J.: HotSpot Internals wiki: CompressedOops, https://web.archive.org/web/20121031151801/https://wikis.oracle.com/display/HotSpotInternals/CompressedOops
13. Sayers, E.W., Barrett, T., Benson, D.A., Bryant, S.H., Canese, K., Chetvernin, V., Church, D.M., DiCuccio, M., Edgar, R., Federhen, S., Feolo, M., Geer, L.Y., Helmberg, W., Kapustin, Y., Landsman, D., Lipman, D.J., Madden, T.L., Maglott, D.R., Miller, V., Mizrachi, I., Ostell, J., Pruitt, K.D., Schuler, G.D., Sequeira, E., Sherry, S.T., Shumway, M., Sirotkin, K., Souvorov, A., Starchenko, G., Tatusova, T.A., Wagner, L., Yaschenko, E., Ye, J.: Database resources of the National Center for Biotechnology Information. Nucleic Acids Research 37(Database issue), D5–D15 (01 2009), http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2686545/
14. Starlight Systems: GNU Trove: High performance collections for Java, https://bitbucket.org/trove4j/trove
15. Tsarkov, D., Vescovo, C.D., Palmisano, I.: Instrumenting atomic decomposition: Software apis for OWL. In: Rodriguez-Muro et al. [11], http://ceur-ws.org/Vol-1080/owled2013_13.pdf
16. Tukaani.org: XZ utils (2015), http://tukaani.org/xz/