Representing and Maintaining Large Corpora

Rüdiger Gleim
Alexander Mehler
Hans-Jürgen Eikmeyer
Department of Computational Linguistics
University of Bielefeld
{Ruediger.Gleim|Alexander.Mehler|hansjuergen.eikmeyer}@uni-bielefeld.de

1 Introduction

The spectrum of electronic corpora which are analysed within the field of corpus linguistics has increased remarkably since the publication of the Brown Corpus (Kucera and Francis, 1967). High efforts of digitizing print media or transcription of speech as well as the limitations of processing speed and storage space made early corpora considerably small in size. With increasing computer performance and cheaper disk storage larger corpora such as the British National Corpus (1991-1994, ~100M words) and the Bank of English Corpus (initiated 1991 by Collins and the University of Birmingham, 524M words) were developed. Today, with the emergence of the World Wide Web and the general trend towards electronic publishing large amounts of text are only few mouse clicks away (Kilgarriff and Grefenstette 2003). On the other hand these sources, first and foremost the WWW pose new challenges to extraction, processing and corpus compilation (c.f. Lüdeling et al. 2006). Nowadays Web based systems for collaborative text production like Wikis invite large communities to actively contribute to web content– thus mixing the classic roles of author and reader. A prominent example is the online encyclopaedia Wikipedia. Beside the tremendous number of articles the high degree of text linkage induced by hyperlinks is remarkable (c.f. Mehler 2006, Zlatic et al. 2006). Therefore we use the Wikipedia as data basis for a touchstone experiment which is described later in this article.

Whereas for annotated text corpora typically trees or treelike representations suffice, other types of corpora demand more sophisticated models. A good example for increasing complexity are multi modal data structures which are studied for instance in research on alignment in communication (Pickering and Garrod 2004). Such approaches of research point towards graph models beyond trees and polyhierarchical extensions thereof (Kranstedt et al. 2007).

The high complexity as well as the sheer size of linguistic resources is a continuous challenge to research on accurate representation models as well as efficient means to handle such data. In this paper we focus on general graph models for representing linguistic data. We introduce a system which allows the creation and maintenance of very large graph structures which can be used to represent virtually any kind of structured data.

In the next section we discuss opportunities and drawbacks of general graph representations. Section 3 introduces the Graph eXchange Language (GXL) (Holt et al. 2006), an XML based language which allows representing graph structures of arbitrary complexity. We use the GXL specification as a basis of our work. Section 4 discusses challenges of large XML based corpora which despite of ever increasing processing power and storage space are still difficult to handle. Section 5 introduces HyGraphDB – a system which aims to combine the comfort and high expressiveness of GXL with efficient means to handle and operate on large graph structures. HyGraphDB is evaluated in section 6 where we use the German distribution of Wikipedia as a demanding test case. The touchstone experiment includes an import of the entire Wikipedia distribution into a HyGraphDB database, the extraction and

insertion of the document interlinking as well as a part of speech tagging of all articles. Section 7 summarizes our findings and gives a prospect of future work.

2 Application specific representation formats vs. general purpose approaches

With the emergence of new research interests and shifting points of view at data, new representation models and -formats as well as tools to work thereon have been developed. The need to share compiled corpora within growing research communities lead to the development of de facto standards which are commonly accepted. Especially XML based languages got popular because of comfort means to specify customized languages for specific purposes. However special research interests inevitably lead to requirements which can not be met *by out of the box* representation formats. To overcome such restrictions extensions of existing formats or entirely new developments have to be found.

General graph based models which abstract from specific applications allow for the representation of arbitrary structured data and are open for extensions. Two prominent examples are Annotation Graphs (Bird and Liberman 2001) and the Linguistic Annotation Framework (Ide and Romary 2004). Furthermore there are approaches which rely on general graph description languages (c.f. section 3, Ide 2007). The key advantage of general graph based approaches is also a major drawback since the way *how* graphs should be used to represent specific data structures can hardly be constrained while keeping full flexibility at the same time. It is thus left to the user (or programmer) how to interpret the data. On the other hand given a graph model (and a representation format) which is expressive enough to subsume arbitrary application specific formats, lossless imports and exports can be realised and the generic graph representation format can thus help to improve interoperability between different applications and proprietary formats by reducing the effort of conversion. In this article we focus on an existing XML based language for graph representation, namely the Graph eXchange Language (Holt et al. 2006).

3 Graph representation for general purpose

The Graph eXchange Language (GXL) was initially designed to enhance interoperability of tools for reengineering in computer science. However it is intended as a graph representation format for general purpose usage and marks the most recent as well as most expressive development. The GXL has emerged from the GRAph eXchange format (Ebert et al. 1999), the Tuple Attribute Language (Holt 1997) and from the format of the PROGRES graph rewriting system (Schürr et al. 1997). The development also orients itself on the specifications of the eXtensible Graph Markup and Modeling Language (Punin et al. 2001) as well as GraphXML (Herman et al. 2000). The GXL consequently avoids application specifics and allows to represent typed, attributed, directed, hierarchical hypergraphs: Nodes can be interconnected either by binary edges or n-ary relations. Furthermore graph elements such as nodes or edges can contain nested sub graphs and thus form hierarchies of arbitrary depth and complexity. Attributes can either contain atomic values of common types such as strings, float or integer values as well as sets or lists. The following XML code demonstrates the graph representation of a sample graph as shown in Figure 1 using GXL.

```
1 <?xml version=”1.0” encoding=”UTF-8”?>
2 <!DOCTYPE gxl SYSTEM ”http://www.gupro.de/GXL/gxl-1.0.dtd”>
3 <gxl>
4   <graph id=”Graph1” edgemode=”directed” hypergraph=”true”>
5     <node id=”A”/>
6     <node id=”B”/>
7     <edge id=”EdgeAB” from=”A” to=”B”/>
8     <rel id=”RelBCD”>
9       <relend dir=”in” target=”B”/>
10       <relend dir=”out” target=”C”/>
```

```
11       <relend dir="out" target="D"/>
12     </rel>
13   </graph>
14   <graph id="Graph2" edgemode="directed" hypergraph="true">
15     <node id="C">
16       <graph id="Graph3">
17         <node id="D"/>
18       </graph>
19     </node>
20   </graph>
21 </gxl>
```

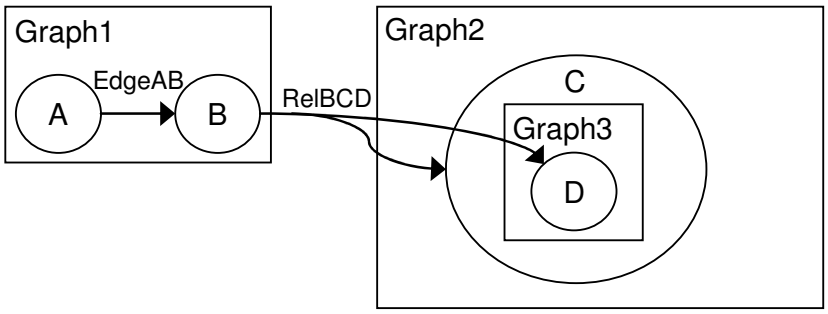

Figure 1 - Hypergraph example

4 Challenges of maintaining large graph structures

The previous section introduced GXL, a XML based language which can be used as means to describe arbitrary graph structures. Even though application specific representation formats have without question their eligibility, general purpose formats such as the GXL can be used as a mediator between different formats. However the broad usability also brings a line of drawbacks. Most important weighs the lack of knowledge of the data specifics and thus means which would allow a specially designed format much more degrees of freedom for access optimization and sparse usage of disk space. Such performance issues can usually be neglected when small amounts of data, such as manually annotated data are processed. But when it comes to large scale automatic annotations of text corpora down to part of speech tagging XML based graph representations tend to explode – compared to the primary data. This holds all the more when a graph representation format such as GXL is being used.

Consider for example the German distribution of Wikipedia, a well-known open encyclopaedia based on collaborative text production. As of 2007-04-27 it consists of 1'559'123 documents (including Portals, Discussion-Pages etc.). The XML dump of the database offered by the Wikipedia Foundation measures about 4.6 GB. However the underlying schema only supports explicit annotation of basic meta data such as document title, timestamp of last contribution and the username or IP-address of the contributor. The aspects which are interesting from a linguistic perspective, such as document interlinking, section structure and not to mention the logical document structure are hidden in the document sources which are stored as a string node along with each document. Parsing and making this information explicit will greatly increase disk space usage (as section 6 will show 134 GB for document interlinking and tagged articles alone). How can such amounts of data be processed in feasible time?

Assuming the XML representation is not going to be revised, there are means which allow relatively good access performance anyway. SAX Parser implementations such as Xerces[1] allow for fast sequential parsing of XML documents. Along with separate indexes performance can further be improved: If the byte offset of desired data is known, retrieval can be performed with almost constant time complexity (i.e. irrespective of the number of

[1] http://xml.apache.org/xerces-c/

documents stored). Consider for example the retrieval of a document by its title. The index necessary to quickly fetch the document from the XML file would have to map from the document title to the corresponding byte offset. Then the SAX Parser can start at that location and only needs to parse the relevant part (i.e. not the entire file from the beginning). This principle is widely used and can be applied to virtually all kinds of retrieval tasks.

A major drawback of the approach described above is that the data is assumed to be static. Edit operations on an XML file are very costly to implement when data is to be inserted or removed. Furthermore the indexes are rendered out of date and need to be rebuilt. So in case of dynamic data structures an alternative solution has to be found.

An obvious option to manage dynamic XML documents are XML capable Database Management Systems such as Tamino, Oracle or eXist. Equivalent to "traditional" relational DBMS they offer concurrent access, transactions, an expressive query language (i.e. XQuery) and can manage any kind of XML document to which a valid schema or DTD exists. Sadly our tests on large GXL documents showed that these systems do not scale well. Often the initial import of a document takes unacceptably long or fails completely. In practice therefore large documents are sometimes split up into logically closed units (e.g. at document level) and inserted separately.

A further principal bottleneck is the access via a query language. On the one hand it offers high expressiveness which comes in handy when complex retrieval tasks have to be performed. On the other hand massive edit operations, for example the insertion of millions of nodes and edges into a GXL document via XQuery include a considerably large overhead for parsing and interpreting the query statements. Furthermore even though a full fledged query language is an elegant way to formulate retrieval tasks, one is fully dependent on how the query processor computes the result. A query processor is (ideally) designed to answer a broad spectrum of typical tasks as efficient as possible. If however a certain type of query is not well supported there are little options left to influence performance.

Our approach aims at representing and handling solely documents which conform to the Graph eXchange Language. By restricting ourselves to one specific schema we can tackle some of the bottlenecks of general purpose XML DBMS by optimizing all access methods for manipulating and maintaining graph structures. The concept and the implementation of the architecture, the HyGraphDB, is described in the next section. Section 6 examines by a series of touchstone experiments based on the German distribution of the Wikipedia in how far the HyGraphDB System can be used for large scale graph structures – and which aspects need further work.

5 HyGraphDB

So far we have put emphasis on XML based languages for representing graph structures, namely the Graph eXchange Language which has the highest degree of expressiveness. Compared to binary formats XML documents tend to be significantly larger and somewhat "wordy". Furthermore, as we have discussed in section 4, there are certain drawbacks regarding maintenance and manipulation. But despite the drawbacks XML has significantly increased interoperability between applications and a world without XML is unthinkable today. HyGraphDB aims to bridge the gap between GXL based graph representation and efficient means to manipulate and access the data. HyGraphDB is developed within the SFB 673 "Alignment in Communication"[2] at Bielefeld University which is funded by the German Research Foundation.

[2] http://www.sfb673.org

HyGraphDB is a programming library for C++ and Java (via Java Native Interface). It offers programmers comfort means to develop applications which work on graph structures without having to bother about the details of the physical storage, concurrency and transactions. Graph structures can either be imported as GXL files (or one of the import filters included so far) or created from scratch within the database using intuitive functions like `add_graph`, `add_node` etc. Lossless exports back to GXL, as well as to various other formats are supported. HyGraphDB itself relies on the BerkeleyDB library which is introduced in the next section. Section 5.2 will then introduce the architecture of HyGraphDB.

5.1 BerkeleyDB

BerkeleyDB is an embedded database library written in C which has initially been developed at the University of California, Berkeley for use in the operating system BSD. An integration within Netscape lead to the foundation of Sleepycat Software which was eventually acquired by Oracle in 2006. In contrast to well known Database Management Systems like MySQL or DB2 BerkeleyDB is not a system which runs out of the box. It can rather be understood as a programming interface which is intended to be used by programmers who have the need to efficiently manage their data without rewriting DBMS-typical functionalities like transactions, locking and physical storage from scratch. So BerkeleyDB can be best thought of as a modular Toolbox which allows realising highly adaptable storage systems. Figure 2 shows the schematic architecture of BerkeleyDB.

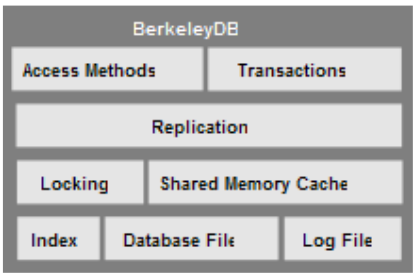

Figure 2 - Architecture of BerkeleyDB (Oracle 2006)

BerkeleyDB supports data to be stored in a flat database model or more technically speaking: key/value pairs of arbitrary byte arrays. The question *how* these byte arrays should be used is entirely left to the programmer as is the configuration of the system modules. The latter includes choices like B-Trees vs. Hashes for data storage, how the physical page and cache sizes should be set and the implementation of sorting functions to order the records. A powerful feature of BerkeleyDB marks the possibility to write custom indexes to speed up retrieval tasks. This is achieved by writing indexing functions that select which database records, and which bits of information thereof should be indexed. That way it is quite easy to write, for example, an index which allows retrieving all nodes of a certain out degree. Summarizing it can be stated that the high degrees of freedom in system configuration are very useful to adapt BerkeleyDB for the specific data at hand – as in our case the representation of graph structures.

5.2 Architecture of HyGraphDB

Programmers interact with graph structures managed by HyGraphDB via a C++ Application Programming Interface. From the user perspective it is the central component of the architecture which is shown in Figure 3. It offers all functions necessary to create, alter and browse attributed and typed hierarchical hypergraph structures conforming to the GXL specification. More precisely: an extension of the GXL is used internally which allows for denser representation of trees by allowing nodes to immediately have ordered nodes as children. In terms of “pure” GXL a tree would have to be represented by explicitly including

the edges. In order to allow the integration of HyGraphDB into Java based applications and client/server systems also a Java API is included and connected via the Java Native Interface (JNI).

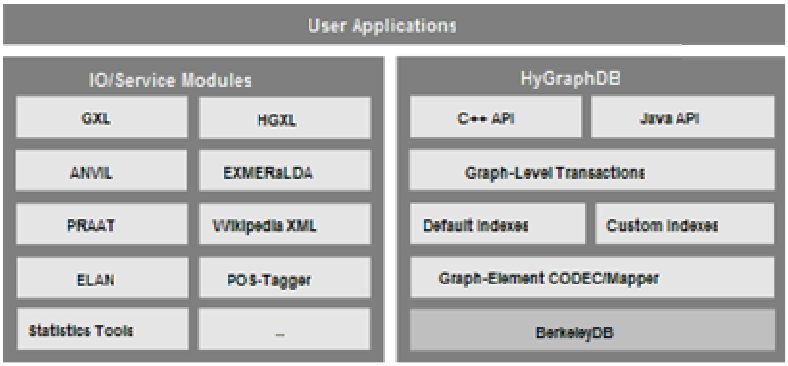

Figure 3 - Architecture of HyGraphDB

Each edit or retrieval operation is automatically protected by an ACID-Transaction (Atomicity, Consistency, Isolation and Durability) to ensure data integrity in case of failure or concurrent access. Furthermore programmers can encapsulate a sequence of access operations into one transaction by beginning and committing a transaction manually. This functionality is internally realised by adapting the BerkeleyDB transaction mechanisms for graph editing. The transaction subsystem can also be switched off entirely if desired to increase throughput. This can be useful for initial imports of large GXL files. Once the file is imported the transaction mechanisms can be switched on again to safely allow multi user access and editing.

Graph elements (i.e. graphs, nodes, binary edges and relations) as well as attributes and values can either be retrieved by using corresponding get-functions or via a query on an index database. The latter returns a list of all data objects which match the specific query. HyGraphDB offers a set of default indexes which can be used, as for example string values or graph elements of a specific type. That way a graph structure which represents a logical document structure for example, can easily be searched for all instances of a specific lemma. This scenario is demonstrated in the evaluations of section 6. Beside the default indexes it is also possible to implement custom indexes by writing a corresponding C++ function.

As already mentioned HyGraphDB relies on BerkeleyDB to store the data and support DBMS functionalities like transactions and locking. Since BerkeleyDB only supports a flat database model an (internal) mapping from graph structures to key/value pairs of byte arrays has to be performed. This is done by the Graph-Element CODEC layer which encodes/decodes each graph element, attribute and value into a database record and back. In order to save disk space we have developed a record level compression algorithm which is integrated into the CODEC-layer as well.

Beside the core API an increasing number of service modules is being developed. Beside the Graph eXchange Language the storage formats of some well known annotation tools are supported, namely ANVIL, ELAN, EXMERaLDA and PRAAT (see Rohlfing et al. 2006 for a survey). Furthermore an import filter for Wikipedia XML dumps has been integrated. The latest addition is the integration of a POS-Tagger which has been developed by Ulli Waltinger at Bielefeld University (www.scientific-workplace.org) to allow programmers seamless POS-tagging within the database. The current development aims at the development and integration of a statistics toolbox to analyse the graph structures stored within the Database.

6 Evaluation

In this section we describe a touchstone experiment in order to examine how well HyGraphDB scales on large graph structures. We choose the German distribution of Wikipedia of 2007-04-27 as a test case. Because of the large number of documents as well as the high degree of document linkage the Wikipedia is a good candidate to explore the capabilities and limits of HyGraphDB.

The system configuration which mainly concerns BerkeleyDB is set as following: Throughout the experiment transactions are switched off. Furthermore the HyGraphDB is configured to maintain an index database to enhance queries for typed string attributes. The primary database is configured to store its data as a B-Tree whereas the string index database shall store its data as a Hashtable. The maximum cache size is set to 8 MB.

6.1 Initial import

In order to get started we need initial data to work on. The MediaWiki Foundation offers XML dumps of its databases which can be downloaded[3] freely. The dumps come in different flavours which vary in how far they cover documents and meta data. We pick the variant "pages-meta-current" which includes all documents in their latest revision. The 4.6 GB (4'934'453'894 Bytes) of XML code contain 1'559'123 documents of which 997'494 are articles. Each document is annotated with its title, information about the latest contribution and most important the MediaWiki source code which is stored within a single string node per document.

The task of the initial import via the HyGraphDB API is to create a graph which contains all documents as nodes. The document nodes are typed according to the Wikipedia specific namespace they belong to (e.g. article, category, image and alike). Furthermore each node is attributed by the title, ID and content. This procedures takes 18.9 min (1'139 s) and consumes 5.6 GB (6'012'350'464 Bytes) of disk space for the primary database and about the same size (5'913'051'136 Bytes) for the index database. The ratio of the size of the primary database file to the imported XML file is about 1.22.

Internally all information about a graph element is stored in *one* record (e.g. IDs of parent element and children, connections, typing etc.). This approach saves retrieval time and storage space compared to alternative approaches where the data fields are distributed over several BerkeleyDB databases (or *tables* in terms of relational database models). The drawback is that the update performance of a graph element has linear time complexity when connections or children are added: Each time a reference to a child or a connection is added, the entire record must be read and rewritten. In practice this behaviour is acceptable and the benefits of improved retrieval times and low memory usage overweigh. However there are application scenarios where hot spots may occur which significantly decrease overall performance. In case of the creation of the document graph it is the very graph element: During the insertions the record of the graph element has to be rewritten as often as document nodes exist. Given 1'559'123 documents and an (uncompressed) space usage of 64 Bit per child reference this sums up to 11.9 MB. To avoid this, the document nodes are not added to the document graph directly but to a set of 1'000 nodes which serve as hubs to better distribute the number of children per element. Because of the element typing this solution does not affect the semantics of the representation essentially.

6.2 Extraction and insertion of document linkage

After the initial import we basically have a graph with a set of isolated document nodes. The next step is to compute the linkage between the documents. The link information is

[3] http://dumps.wikimedia.org/

hidden in the document source code which has been stored as string attributes of the document nodes as part of the import. Therefore the sources of each document have to be parsed, the contained link information extracted and the binary edges to the specific target document nodes to be inserted. Internally not only the edges are inserted but the connected nodes also need to be updated. Since the information of the target node may be located virtually at any location within the primary database this procedure is a first test for the lookup performance of graph elements via keys through the internal B-Tree.

The task takes 2.96 hours (10'641 s) to compute. The representation of the linkage within the database consumes additional 3,1GB (3'300'659'200 Bytes) compared to the initial import. This means an average memory usage of 129.2 Bytes per inserted edge which appears to be quite a lot. A good part of this amount is not consumed by the record data (i.e. user data) but by B-Tree overhead which BerkeleyDB uses for storage. A Hashmap would have been more space efficient in this case but at cost of less throughput during insertion and browsing through the graph structures.

How does the throughput of the system in terms of processed links develop over time? Figure 4 shows the number of extracted and inserted links per ms over the number of processed documents. The throughput appears to be relatively stable, even though a slight decrease can be observed. This decrease is likely due to the massive updates of the database records which represent the connected nodes: Each time a node is connected by another edge or relation the respective record is updated and extended by a few bytes to store linkage information. With an increasing number of connections per node the time needed to update the record increases slightly so the overall throughput drops respectively. However regarding the overall performance in context of 25'542'909 links being inserted it is acceptable.

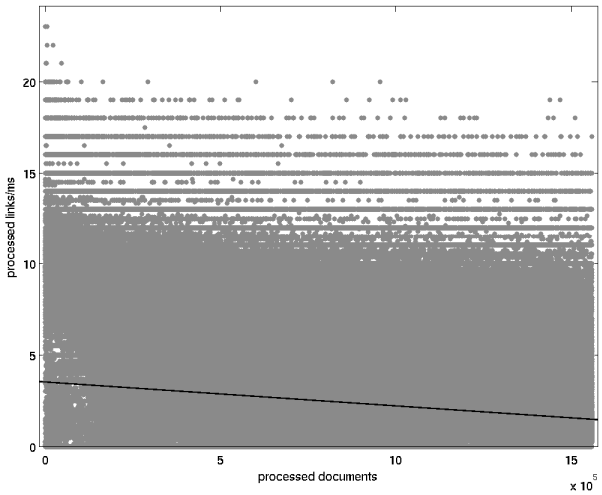

Figure 4 - Processed links/ms over the number of parsed documents

6.3 Part of speech tagging

At this point the database contains a graph of highly interconnected document nodes. However the document structures are still hidden in the "flat" string attributes which store the documents contents. Therefore the next, and certainly most demanding step of the experiment is to perform a part of speech tagging in the HyGraphDB graph database and to consequently store the resulting document structures as a typed and attributed graphs. The document graphs are stored as children of their corresponding document nodes. Each logical element, from headings and paragraphs over sentences down to token level is explicitly represented by a typed node (see Figure 5 for an example). The word form and the lemma of each token node

are stored as typed string attributes. We restrict the task to articles (i.e. 997'494 documents are processed, skipping discussion pages etc.).

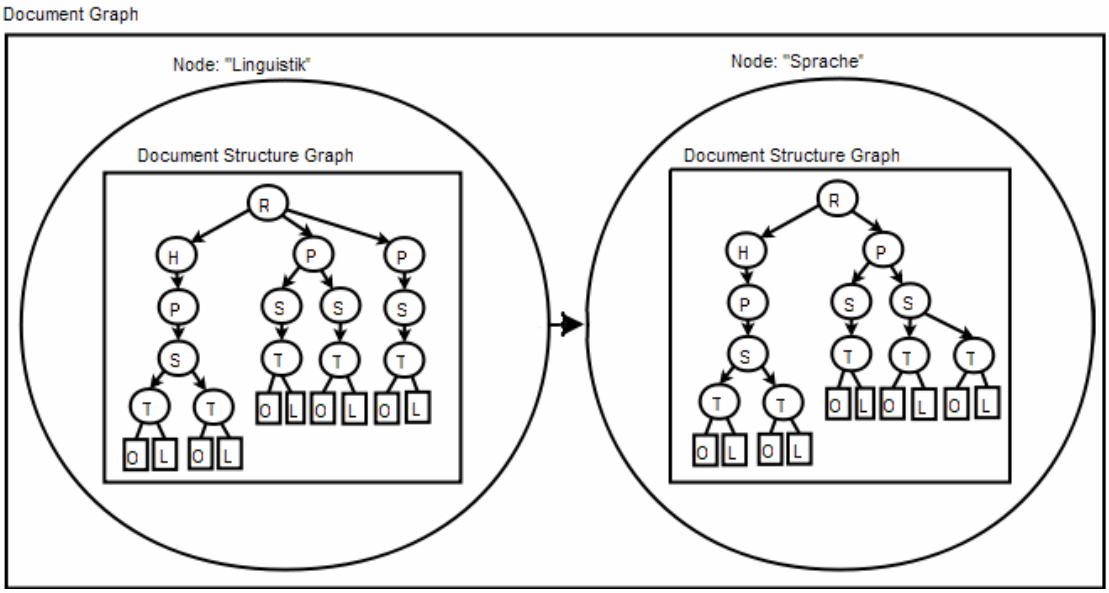

Figure 5 - Graph representation of Wikipedia (example)

The overall process of tagging takes 31.1 hours (112'038s) which marks an average processing time of 0.112s per article. Figure 6 shows the throughput of the tagging and LDS tree creation by processed characters/ms over the number of processed documents. The plot shows an overall constant behaviour. Compared to the size of the primary database of the linked documents additional 75.8 GB (81'435'684'864 Bytes) are needed as well as additional 8.2 GB (8'816'619'520 Bytes) for the index database. This leads to a total of 85 GB in the primary database as well as 14 GB in the index database to store the German distribution of Wikipedia with complete linkage and POS-tagged articles as a graph structure.

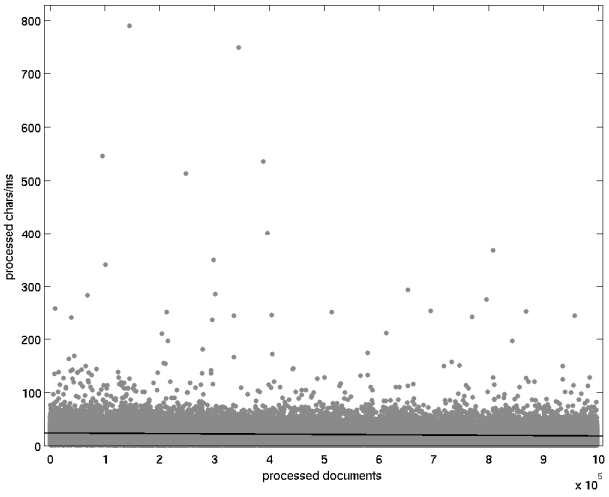

Figure 6 - Processed characters/ms over number of parsed documents

6.4 Rank frequency distribution of document in/out degrees

Now with the Wikipedia being imported, the link structure extracted and the LDS annotated the subsequent tests aim at data analysis. The first task is to compute the rank

degree distribution of the in/out degrees of the document nodes. In order to complete this task each node of the document graph has to be inspected and the information about connected edges to be extracted.

The task takes 2'460s (i.e. 1.58 ms/document) to compute the distributions. Figure 7 shows a log-log plot of the distributions – distinguished into total degree (solid line), in degree (dotted line) and out degree (dashed line).

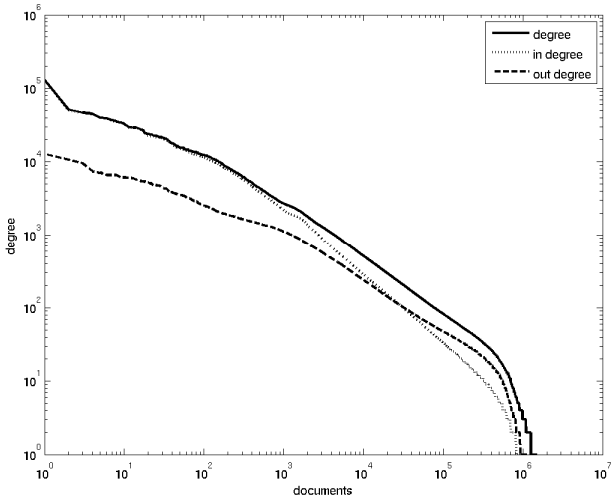

Figure 7 - Distribution of document node degrees

6.5 Rank frequency distribution of lemmas

In the next step of the evaluation experiment the rank frequency distribution of all lemmas within the graph database is computed. This is accomplished by iterating over all entries of the primary database (using a database cursor) and fetching out all attribute elements which represent a lemma. Theoretically it would be more efficient to iterate over the index database which solely contains references to the string attributes (among others the desired lemmas). However due to the internal storing techniques of BerkeleyDB and their configuration via HyGraphDB in its current state it is faster to accomplish the task via the primary database: A database cursor which iterates over the primary database achieves a throughput of ~24 MB/s whereas a cursor over the index database can only read at about 700 KB/s.

The task takes 61.3 minutes (3'676s) to complete. The log-log plot of the lemma frequencies as shown in Figure 8 reveals a power law like distribution. Other than expected by Mandelbrot we observe a very good fitting in terms of Zipfs first law (Tuldava 1998).

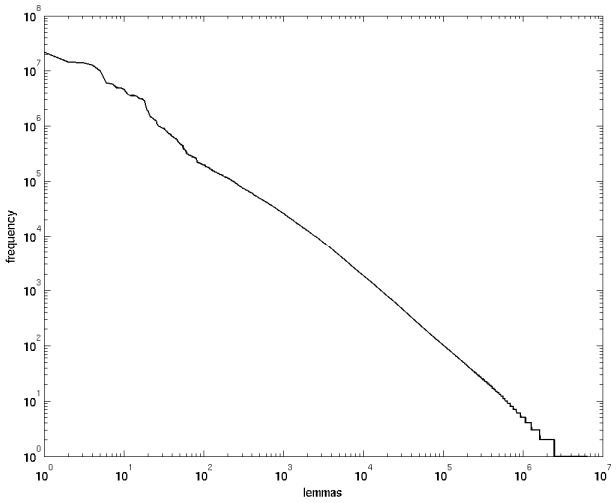

Figure 8 - Distribution of lemmas

6.6 Index based query for lemmas

In this section we test the performance of index based queries. More precisely, we are interested in how well queries for all instances of a specific lemma scale with their frequency. The previous experiment already returned the rank frequency distribution of lemmas. Based on this distribution we pick sample lemmas with a frequency according to a power of two (or as close as possible to a power of two) up to the most frequent lemma "der" which appears 21'523'092 times. Then we perform index based queries in order to find all instances and measure the time needed.

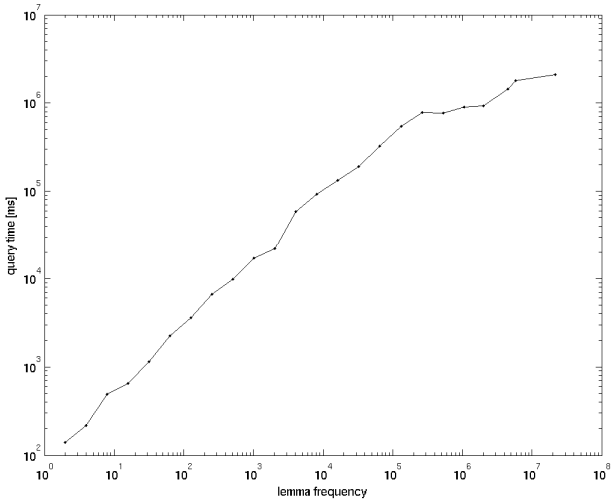

Figure 9 - Query performance

The log-log plot of the query times needed to retrieve all the instances of a selected lemma (see Figure 9) demonstrates a power law behaviour (with a positive exponent). This means good retrieval performance for most cases – on the other hand the retrieval time increases according to a power law and, thus, significantly, in the case of the most frequent lemmas. Therefore, retrieval tasks which require the lookup of a large number of lemmas better iterate sequentially over the primary database as done in the previous step rather than to look up the lemma occurrences via the index.

6.7 Export

To conclude the line of experiments we perform an export of the entire graph structure within the database into a HGXL document, a GXL extension which allows for a more compact representation of trees. The export measures 134 GB (143’507’923'859 Bytes) and thus exceeds the size of the database file (85 GB) by far. The export procedure took 28.66 hours (103'176 s) which marks an average throughput of 1.33 MB/s. Apparently the export function needs further improvement.

7 Conclusion

The article introduced HyGraphDB, a system which allows the representation and handling of complex and large hypergraph structures. The evaluation experiments showed that the approach can also be used for large data quantities such as Wikipedia. However the tests also revealed aspects which need further improvement. This regards especially graph elements with a very high number of children or connections which can have a critical impact on performance as discussed in section 6.1. Furthermore the configuration of the index databases needs further evaluation in order to achieve higher throughput.

Since HyGraphDB is still under development it is not yet freely available. However researchers are welcome to use HyGraphDB via a web-based corpus management system. In the near future, the system will offer guests access to corpora made public by the SFB 673 and will also offer other resources as, for example, annotated Wikipedia releases.

Acknowledgement

The HyGraphDB is developed within the research project “*Multimodal alignment corpora: statistical modeling and information management*”[4] as part of the SFB 673 “*Alignment in Communication*” which is funded by the *German Research Foundation* (DFG).

Bibliography

Bird S. and M. Liberman (2001) A formal framework for linguistic annotation, Speech Communication 33(1,2), pp 23-60

Ebert J., B. Kullbach, and A. Winter (1999). GRAX – An Interchange Format for Reengineering Tools. In SixthWorking Conference on Reverse Engineering, pages 89–98, Los Alamitos, IEEE Computer Society.

Herman I. and M. Scott Marshall (2000). GraphXML - an XML-based graph description format. In Graph Drawing, pages 52–62.

Holt R. (1997) TA: The tuple attribute language. http://plg.uwaterloo.ca/ holt/papers/ta-intro.htm, (21.08.2006).

Holt R. C., A. Schürr, S. Elliott Sim, and A. Winter (2006). GXL: A graphbased standard exchange format for reengineering. Science of Computer Programming, 60(2):149–170.

Ide N., L. Romary (2004). International standard for a linguistic annotation framework. Journal of Naturaql Language Engineering, 10:3-4, 211-225.

Ide N. (2007). Annotation Science: From Theory to Practice and Use. Data Structures for Linguistic Resources and Applications. Proceedings of the Biennial GLDV Conference 2007. Tübingen: Narr.

[4] http://www.sfb673.org/projects/X1/

Kilgarriff, Adam and Gregory Grefenstette (2003) Introduction to the special issue on the Web as corpus. In: Computational Linguistics 29(3), 333-347

Kranstedt, A., A. Lücking, A. Mehler, T. Pfeiffer, and H. Rieser (2007). A multimodal corpus for speech and pointing gestures. Submitted.

Kucera, H. and W. N. Francis (1967) Computational Analysis of Present-Day American English. Brown University Press, Providence, RI, USA.

Lüdeling A., S. Evert and M. Baroni (2006) 'Using web data for linguistic purposes.' Language and Computers, 7-24

Mehler, A. (2006). Text linkage in the wiki medium – a comparative study. In Karlgren, J., editor, Proceedings of the EACL Workshop on New Text – Wikis and blogs and other dynamic text sources, April 3-7, 2006, Trento, Italy, pages 1–8.

Oracle (2006). A Comparison of Oracle Berkeley DB and Relational Database Management Systems, http://www.oracle.com/database/docs/Berkeley-DB-v-Relational.pdf

Pickering, M. J. and S. Garrod (2004). Toward a mechanistic psychology of dialogue. Behavioral and Brain Sciences, 27:169–226.

Punin J. and M. Krishnamoorthy (2001). XGMML (extensible graph markup and modeling language). http://www.cs.rpi.edu/ puninj/XGMML/. (21.08.2006).

Schürr A., A.J. Winter, and A. Zündorf (1999). PROGRES: Language and Environment, volume 2, pages 487–550. World Scientific, Singapore.

Rohlfing K., D. Loehr, S. Duncan, A.Brown, A. Franklin, I. Kimbara, J.-T. Milde, F. Parrill, T. Rose, T. Schmidt, H. Sloetjes, A. Thies and S. Wellinghoff (2006). Comparison of multimodal annotation tools. Gesprächsforschung, 7:99–123.

Tuldava J. (1998). Probleme und Methoden der quantitativ-systemischen Lexikologie, Wissenschaftlicher Verlag, Trier

Zlatic, V., M. Bozicevic, H. Stefancic, and M. Domazet (2006). Wikipedias: Collaborative web-based encyclopedias as complex networks.