Storing UIMA CASes in a relational database

Georg Fette[1 2], Martin Toepfer[1], and Frank Puppe[1]

[1] Department of Computer Science VI, University of Wuerzburg,
Am Hubland, Wuerzburg, Germany
[2] Comprehensive Heart Failure Center, University Hospital Wuerzburg,
Straubmuehlweg 2a, Wuerzburg, Germany

Abstract. In the UIMA text annotation framework the most common way to store annotated documents (CAS) is by serializing the document to XML and storing this XML in a file in the file system. We present a framework to store CASes as well as their type systems in a relational database. This does not only provide a way to improve document management but also the possibility to access and manipulate selective parts of the annotated documents using the database's index structures. The approach has been implemented for MSSQL and MySQL databases.

Keywords: UIMA, data management, relational databases, SQL

1 Introduction

UIMA [2] has become a well known and often used framework for processing text data. The main component of the UIMA infrastructure is the CAS (Common Analysis Structure), a data structure which combines the actual data (the text of a document), annotations on this data and the type system the annotations are based on. In many UIMA projects CASes are stored as serialized XML-files in file folders with the corresponding type system file in a separate location. In this storage mode the resource management to load which CAS with which type system lies in the responsibility of the programmer who wants to perform an operation on specific documents. However, manual management of files in folders on local machines or network folders can quickly become confusing and messy especially when projects get bigger. We present a framework to store CASes as well as their corresponding type systems in a relational database. This storage mode provides the possibility to access the data in a centralized, organized way. Furthermore the approach provides all benefits that come along with relational databases including search indices on the data, selective storage, retrieval and deletion as well as the possibility to perform complex queries on the stored data in the well known SQL language.

The structure of the paper is as follows: Section 2 describes the related work, Section 3 describes the technical details of the database storage mechanism, Section 4 illustrates query possibilities using the database, Section 5 demonstrates performance experiences with the framework and Section 6 concludes with a summary of the presented work.

2 Related Work

The only approach known to the best knowledge of the authors where CASes are stored in a database is the Julielab DB Mapper [4] which serialized CASes to a PostgreSQL database. However, the mechanism does not store the CASes' type systems nor does it support features like referencing of annotations by features or derivation of annotation types. Other approaches use indices to improve query performance but do not allow to reconstruct the annotated documents from the index (Lucene based: LUCAS [4], Fangorn [3]; relational database based: XPath [1], ANNIS [7]; proprietary index based: TGrep/TGrep2 [6], SystemT [5]). The indices still need the documents to be stored in the file system. Furthermore some of the mentioned indices only allow specialized search capabilities (e.g. emphasis on parse trees) which are provided by the respective search index and cannot search directly on the UIMA data structures. In contrast to these approaches our system allows searches on arbitrary type systems by formulating queries closely related to the involved annotation and feature types.

3 Database storage

The storage mechanism is based on a relational database for which the table model is illustrated in Figure 1. The schema can be subdivided in a document related part (left), an annotation instance part (middle) and a type system related part (right). Documents are stored as belonging to a named collection and can be manipulated (retrieved, deleted, etc.) as a group, e.g. deleting all annotations of a specific type. Annotated documents can be handled individually by loading/saving a single CAS or by processing a whole collection by creating a collection reader/writer. In either way any communication (loading/saving) can (but need not) be parametrized so that only desired annotation types are loaded/saved, thus speeding up processing time, reducing memory consumption and facilitating debugging processes. A type system, instead of being stored in an XML file and containing a fixed type system, can be retrieved from the database in different task specific ways. One way is by requesting the type system which is needed to load all the annotated documents belonging to a certain collection. Other possibilities are by providing a set of desired type names or by providing a regular expression determining all desired type names. The storage mechanism is able to store the inheritance structures of UIMA type systems as well as referencing of annotations by features of other annotations. For further information on the technical aspects we refer to the documentation of the framework[3].

4 Querying

A benefit from storing data in an SQL database is the database index and the well established SQL query standard. The database can be queried for counts of occurrences of specific annotation types, counts of covered texts of annotations or even complex annotation structures in the documents. We want to exemplify this with a query on documents which have been annotated with a dependency parser using the type system shown in Figure 2.

[3] http://code.google.com/p/uima-sql/

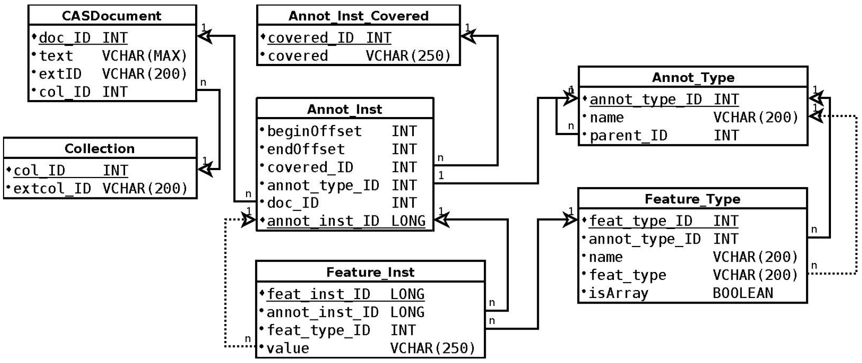

Fig. 1. schema of the relational database storing CASes and type systems.

```
<typeDescription>
<name>Token</name>
<supertypeName>
uima.tcas.Annotation
</supertypeName>
<features>
<featureDescription>
<name>Governor</name>
<rangeTypeName>Token</rangeTypeName>
</featureDescription></features>
</typeDescription>
```

Fig. 2. type system for parses

```
SELECT govText.covered FROM
annot_inst govToken, annot_inst_covered govText,
annot_inst baseToken, annot_inst_covered baseText,
feat_inst, feat_type WHERE
baseText.covered = ’walk’ AND
baseToken.covered_ID = baseText.covered_ID AND
baseToken.annot_inst_ID = feat_inst.annot_inst_ID AND
feat_inst.feat_type_ID = feat_type.feat_type_ID AND
feat_type.name = ’Governor’ AND
feat_inst.value = govToken.annot_inst_ID AND
govText.covered_ID = govToken.covered_ID
```

Fig. 3. SQL query for governor tokens

To query for all words governing the word *walk*, we have to look for tokens with the desired covered text, find the tokens governing those tokens and return their covered text. The SQL command for this task is shown in Figure 3. An abstraction layer to cover the complexity could be put on top (like a graph querying language), but even in the presented way with standard SQL the capabilities of the database engine can serve as a useful tool to improve corpus analysis.

5 Performance

To run a performance test on the storage engine we created a corpus of 1000 documents, each consisting of 1000 words. The words were taken from a dictionary of 1000 randomly created words, each of 8 characters length. From each document we created a CAS and added annotations so that each word was covered, with the annotations covering 1 to 5 successive words. Each annotation was given two features, one String feature with a value randomly taken from the word dictionary and a Long feature containing a random number. All documents were stored and then loaded again. This was done with the database engine as well as with a local file folder on the same hard drive the database files were located on. In a second experiment the same documents where loaded again and we added an annotation of another type with a Long feature containing a random number to each document. After adding the additional annotation the documents were stored again. In a third experiment we wanted to query for the frequencies of annotations covering each of the words from the word dictionary.

For file system storage this was done by accumulating the annotation counts during an iteration over all serialized CASes, for database storage this was done by performing a single SQL query for each of the words from the dictionary.

In Table 1 we can observe that the time needed for database storage is quite long but reading is as fast as from the file system. Storing to the database during the second experiment was faster than in the first one, because this time only the additional annotations had to be incrementally stored. Storage to the file system again performed about five times faster than to the database but the benefit of being able to incrementally store only the additional annotations can be clearly observed. Physical storage space consumption is larger for database storage but that shouldn't pose a major problem as hard disc space is not an overly expensive resource nowadays. Query performance in the database is about 20 times faster than using file system storage illustrating the benefit of the database approach.

Table 1. Performance measures comparing database and file system storage

	exp1		exp2		exp3
	saving (sec.)	loading (sec.)	saving (sec.)	storage size (MB)	query (sec.)
DB	36.0	1.1	7.2	42.3	0.16
FileSystem	2.6	1.1	2.7	6.5	7.0

6 Conclusion

We have presented a framework to store/retrieve CASes and perform analysis queries on them using a relational database. We examined the save, load and query speed compared to regular file based storage and presented examples how to use the database index structures to analyze annotations in the corpus. We hope to be able to improve the storage speed of the database engine so that the choice between file system storage and database storage will not be influenced by the still quite large difference in speed performance.

This work was supported by grants from the Bundesministerium fuer Bildung und Forschung (BMBF01 EO1004).

References

1. Bird, S., Lee, H.: Designing and evaluating an xpath dialect for linguistic queries. In: 22nd International Conference on Data Engineering (2006)
2. Ferrucci, D., Lally, A.D.A.M.: Uima: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering 10(3-4), 327–348 (2004)
3. Ghodke, S., Bird, S.: Fangorn: A system for querying very large treebanks. In: COLING (Demos). pp. 175–182 (2012)
4. Hahn, U., Buyko, E., Landefeld, R., Mühlhausen, M., Poprat, M., Tomanek, K., Wermter, J.: An overview of JCoRe, the JULIE lab UIMA component repository. In: LREC'08 Workshop 'Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP' (2008)
5. Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F., Vaithyanathan, S., Zhu, H.: Systemt: a system for declarative information extraction. SIGMOD Rec. (2009)
6. Rohde, D.L.T.: Tgrep2 user manual (2001)
7. Zeldes, A., Lüdeling, A., Ritz, J., Chiarcos, C.: Annis: a search tool for multi-layer annotated corpora. In: Proceedings of Corpus Linguistics 2009 (2009)