Supporting Annotation Layers for Natural Language Processing

Archana Ganapathi and **Preslav Nakov** and **Ariel Schwartz** and **Marti Hearst**
CS Division and SIMS
University of California, Berkeley
{*archanag,nakov,sariel*}*@cs.berkeley.edu, hearst@sims.berkeley.edu*

Abstract

This paper proposes a simple mechanism for supporting multiple overlapping layers of annotations for natural language text processing. We present a query language for flexibly accessing annotated text, and demonstrate its use on examples taken from the NLP literature. We then report on experiments comparing different storage and indexing architectures using an RDBMS, showing that annotation-based queries can be made to scale to large corpora with many layers of annotations.

1 Introduction

Today most natural language processing (NLP) algorithms make use of the results of previous processing steps. For example, a word sense disambiguation algorithm may use the output of a tokenizer, a part-of-speech tagger, a phrase boundary recognizer, and a module that classifies noun phrases into semantic categories. Currently there is no standard way to represent and store the results of such processing for efficient retrieval.

In this paper we propose an annotation framework for marking text up with processing results, a query language for flexibly accessing portions of text that have been so annotated, and indexing architectures for efficiently performing retrievals against the annotated text. The model allows for both hierarchical and overlapping layers of annotation as well as for querying at multiple levels of description. We demonstrate the power of the query language and the efficiency of the indexing architecture on a wide variety of query types that have been published in the NLP literature.

The architecture is built on top of a relational database management system (RDBMS), and so can take advantage of advanced indexing structures supplied by such systems. We believe this paper is the first to experiment with different indexing structures in order to determine how to make annotation-based queries scale to very large corpora with many layers of annotations.

An added advantage of this representation is that it supports an infrastructure for running computational linguistics experiments. Often NLP algorithms require sampling from a document collection, either to ensure that training and testing are conducted over representative data, or to find sufficient numbers of sentences that satisfy certain properties. For example, a disambiguation algorithm may need to train on examples of a word within various domains, since its meanings may vary across different subcollections.

In the remainder of the paper we describe related work, illustrate the annotation model and the query language on examples drawn from the NLP literature, describe different indexing architectures and the experimental results, thus showing the feasibility of the approach for a variety of NLP tasks.

2 Related Work

(Bird and Liberman, 2001) and (Cassidy and Bird, 2000) provide excellent analyses of the issues surrounding annotation of speech and text collec-

Submitted for publication, Feb 2004

tions, and (Bird and Harrington, 2001) summarize a recent special journal issue on the topic.

(Bird and Liberman, 2001) introduce an abstract general approach, based on *annotation graphs* (AG): partially ordered directed acyclic graphs with labelled arcs and nodes of positive degree. Demonstrated on speech data, most nodes have time stamps or are constrained via paths to labelled predecessors and successors. AGs quickly gained popularity and different tools have been developed (Bird and Liberman, 2001), including AGTK[1] by the Linguistic Data Consortium (Xiaoyi et al., 2002). The AGTK database model is complex: it consists of 7 tables plus an additional one for each corpus.

There have been some theoretical (Bird and Liberman, 2001) and practical (Bird and Harrington, 2001; Xiaoyi et al., 2002) attempts to implement a query language for AGs. A sample query that finds arcs labelled as words, whose phonetic transcription starts with a '*hv*' is:

```
SELECT I
WHERE X.[id:I].Y <- db/wrd
      X.[:hv].[]*.Y <- db/phn;
```

AGTK supports this query language by mapping it to SQL (Xiaoyi et al., 2002). To support the fact that arc tracing can go in arbitrary directions, all pairs of connected nodes are pre-computed and stored in a table, which can be space intensive.

The Emu speech database management system (Cassidy and Harrington, 2001) supports *sequential* levels of annotations, within which each *token* can have one or more *labels* and is *optionally* time labeled. Hierarchical relations may exist between tokens in different levels, but must be explicitly defined for each pair. The basic Emu query language elements include: label disjunction, comparisons, relative position (e.g. *start*), sequence(->) and dominance (^) operators. E.g. the query

```
[[Phonetic=A -> Phonetic=p] ^
  Syllable=S]
```

matches sentences of phonetic '*A*' followed by '*p*' both dominated by an '*S*' syllable. Although designed to be compatible with the relational model, the implementation in (Cassidy and Harrington, 2001) performs a linear search of the database in order to answer queries. (Cassidy, 1999) describes converting Emu queries into SQL. The core of the model is a table with document ID, level, start time, end time, sequence (within the level), utterance and label columns. Timing experiments were performed on a collection of 1,000 sentences using simple queries (e.g., looking for one or two tokens spanned by a third) and significant speedup was found as compared to linear search.

The Q4M query language for the MATE annotation workbench (McKelvie et al., 2001; Mengel et al., 1999) also uses a directed graph model, and is highly expressive, allowing for specification of constraints and ordering in the annotated components. However, the system uses XML for storage and retrieval, and is currently implemented with an in-memory representation, thus calling into question its scalability. The following query finds nouns which are followed by the word *lesser*:

```
($a word) ($b word); ($a pos ~ "NN")
  && ($a <> $b) && ($b # ~ "lesser")
```

The Tagged Information Management System (TIMS) (Nenadic et al., 2002), defines the TIQL language, where queries consist of manipulating intervals of text, indicated by XML tags, using set operations (intersection, union, difference, concatenation). For example,

```
(<SENTENCE> ⊕ <TERM nf='COUP TF II'>) ⊕
 <V lemma='inhibit'>
```

specifies the sentences containing both the noun phrase *COUP-TF II* and the verb *inhibit*. It was evaluated on fewer than 6,000 annotations yielding performance similar to that of regular expressions. The TIMS implementation stores tag information separately from the documents in a table with start and end positions, tag type and layers.

3 The Layered Query Language

Our approach emphasizes efficient and flexible querying and retrieval, while most other recent work on speech and text annotation tends to focus on the manual annotation process. Furthermore, we assume that the underlying text is fairly static, so we can express the annotations in terms of absolute character positions (whereas in AGs the assumption is that the start and end time of the events may be unknown, but limited within a time interval). This in turn allows storage of annotations within just one table containing *document*

[1] Annotation Graph Toolkit: http://agtk.sourceforge.net

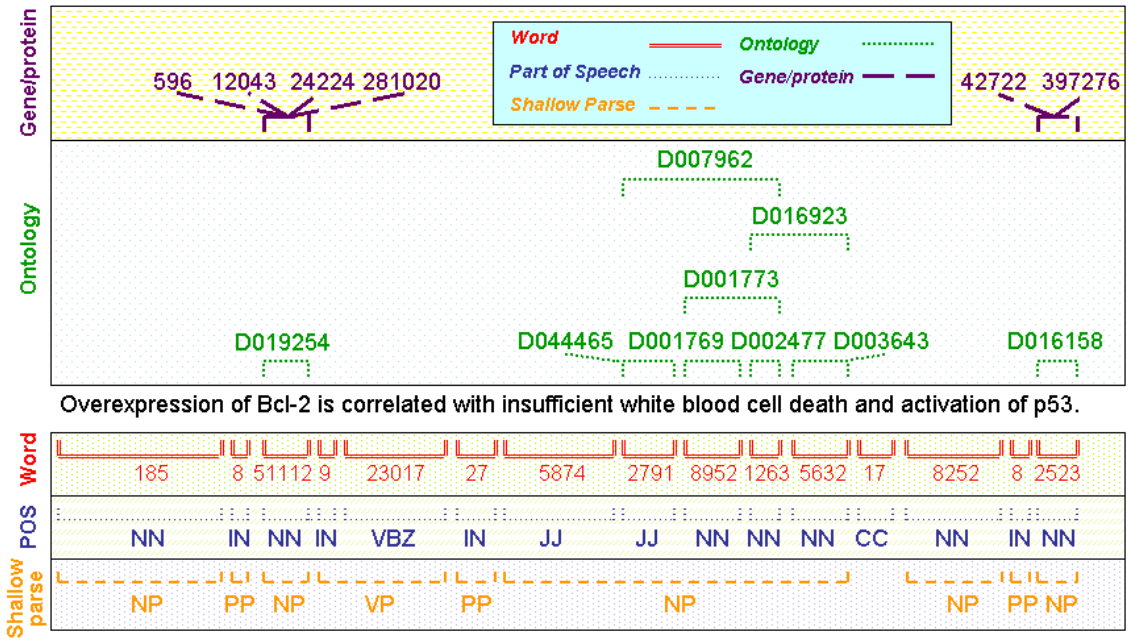

Figure 1: Illustration of the annotation layers. The *full parse*, *sentence* and *section* layers are not shown.

id, *tag type*, *start* and *end* positions for each annotation. Hence we do not optimize for efficient editing but focus on compact representation, easy query formulation, addition and removal of layers, and straightforward translation into SQL.

Below we illustrate our XML-like *Layered Query Language (LQL)*. These queries are drawn from published research, to ensure the real-world applicability of the annotations and experiments described below. Although we use examples from bioscience NLP, the features are useful for many NLP tasks, regardless of domain.

3.1 Protein-Protein Interaction

An important open question in molecular biology is: which proteins interact with which other proteins? Recently researchers have discovered that the bioscience journal collection MEDLINE[2] can be mined to extract hypotheses about such relations. One approach is to begin with a list of protein names, and look for sentences that contain two or more of them. This is complicated by the fact that there are multiple ways to represent the name of a protein, and additionally, proteins are often expressed as either multi-word terms or acronyms. Thus a pre-processing step is usually run to label terms that correspond to protein names.

Figure 1 illustrates how the layered annotation framework would support the storage of a sentence that is relevant for such a query. Each annotation represents an interval spanning a sequence of characters, using absolute start and end positions. Each layer corresponds to a conceptually different kind of annotation (e.g., word, gene/protein[3], shallow parse). Layers can be *sequential*, *overlapping* (e.g., two multiple-word concepts sharing a word) and *hierarchical* (either in terms of *spanning*, when the intervals are nested as in a parse tree, or *ontologically*, when the token itself is derived from a hierarchical ontology).

There is a one-to-one correspondence between the word and the part-of-speech (POS) layers. The word, POS and shallow parse layers are *sequential* (the latter can skip words or span multiple words). Assuming a pre-processing step has identified which stretches of text can be associated with a gene/protein name, the gene/protein layer shows the results of assigning IDs from the LocusLink[4] database of gene names. This layer as-

[2] http://www.nlm.nih.gov/pubs/factsheets/medline.html

[3] Genes and their corresponding proteins often share the same name and the difference between them is often elided.

[4] http://www.ncbi.nlm.nih.gov/LocusLink

signs to each gene as many LocusLink IDs as the number of organisms having the gene (four in the case of Bcl-2 and two in the case of p53).

Another preprocessing step has assigned terms from the hierarchical medical ontology MeSH[5] (Medical Subject Headings). Note that these are overlapping (share the word *cell*) and hierarchical: both *spanning*, since *blood cell* (with MeSH ID D001773) is a type of *cell*, and *ontologically*, since *blood cell* is a kind of *cell* and *cell death* (D016923) is a type of *Biological Phenomena*.

Returning now to the protein-protein interaction example, there is no guarantee that mere co-occurrence of gene/protein names implies a true interaction. To help produce more accurate results, (Blaschke et al., 1999) developed a list of verbs (and their derived forms) and scanned for sentences containing the pattern PROTEIN ... INTERACTION-VERB ... PROTEIN. This query, which we label (a) for later reference, can be expressed in LQL as follows:

```
<document
   <sentence
      <gene_protein> {print tag_type}
      ...
      <pos [tag_type=verb]
         <word [lex=correlated]> >
      ...
      <gene_protein> {print tag_type}
   > {print}
> {print document.id}
```

This query retrieves all sentences with a protein name in the gene/protein layer, followed by any sequence of words, followed by the interaction verb "correlated", followed by any sequence of words, and finally by another protein name from the gene/protein layer. Then it outputs the LocusLink IDs of the two proteins, the text of the sentence they have been found in, and the ID of the corresponding document.

Each level of the query states the layer it refers to (sentence, part-of-speech, gene/protein) followed optionally by restrictions on the attribute values, enclosed in square brackets, to distinguish tokens within the layer. This is followed optionally by an action statement, enclosed in curly braces, which either binds variables or specifies which parts of the matched pattern should be printed. The ellipses (...) indicate that tokens may intervene in between the specified patterns. Note that within square brackets, in addition to equality, the language allows inequality (`!=`), conjunction (`&&`), and disjunction (`||`). Thus the query above can be modified to search for more than one verb simultaneously by writing `<word [lex=activates|inhibit|binds]>`. Algebraic comparisons for the *length* attribute (e.g., `<word [length<=5]>`), are also possible.

All layers have a `lex` (the text spanned by the corresponding interval) and a `tag_type` (e.g., *verb* for the POS layer) attribute. When invoked with no arguments, `print` outputs the value of the `lex` attribute. We can refer to an attribute directly (e.g., `print tag_type`) or with a fully qualified name (e.g., `print document.id`).

(Thomas et al., 2000) also tackle the protein-protein interaction problem using contextual rules with complex syntactic structure, e.g., `NP VERB PREPOSITION NP` (with no intervening tokens). They investigated over 30 verbs for this task, but found only 3 yielded high-quality results consistently: *interact (with)*, *associate (with)* and *bind (to)*. One of their queries can be expressed in LQL as follows: (b)

```
<sentence
  <shallow_parse [tag_type=NP]>{$np1=lex}
  <pos [tag_type=verb] <word [lex=binds]>>
  <pos [tag_type=prep] <word [lex=to]>>
  <shallow_parse [tag_type=NP]>{$np2=lex}
> {print $np1, $np2, sentence}
```

3.2 Descent of Hierarchy

(Rosario et al., 2002) hypothesize that one can predict which semantic relation holds between a pair of words in a noun-noun compound based on the MeSH subhierarchies the words are assigned to.[6] For example, a particular relation holds for two-word noun compounds for which the first word falls within the A01 (*Body Regions*) subhierarchy and the second one falls within A07 (*Cardiovascular System)*. Sample noun compounds include *shoulder artery*, *limb vein* and *forearm microcirculation*. Within MeSH the notation is arranged hierarchically, so that, for example, A07 maps to *Cardiovascular System*, A07.231 is *Blood Vessels*, A07.231.114 is *Arteries*, and A07.231.114.056 is

[5] http://www.nlm.nih.gov/mesh/meshhome.html

[6] In Figure 1 a different MeSH representation appears.

Aorta. Thus to retrieve all terms in the subhierarchy of A07, we simply search for those terms whose code begins with the string "A07". The LQL query for this example is as follows: (c)

```
<shallow_parse [tag_type=NP]
   <pos [tag_type=noun]
     <MeSH [label=A01*]>{print}>
  <pos [tag_type=noun]
     <MeSH [label=A07*]>{print}> $>
```

The two special characters ^ and $ are used as in Unix regular expressions to match the parent term's beginning and end positions, respectively. Note that the $ indicates that the two nouns should occupy the last two positions inside the NP. The * is used as in standard regular expressions; in this case, it matches all MeSH *labels* (using the `label` attribute; different from `tag_type`, which is MeSH ID) that fall within the A01 and A07 subhierarchies, respectively. This query will retrieve noun compounds with no preceding modifiers.

3.3 Acronym-Meaning Pair Extraction

(Pustejovsky et al., 2001) describe a system for extracting acronym-meaning pairs such as in "*We describe a system for Automatic Speech Recognition* (*ASR*)." to find the relation *Automatic Speech Recognition* $\Leftrightarrow$ *ASR*. They process the text with a shallow parser and then look for syntactic patterns, e.g., "`NP (NP)`", "`NP, NP`" and "`(NP) NP`". The LQL formulation for the first one is: (d)

```
<shallow_parse [tag_type=NP]>{print}
<pos [tag_type=\(]>
<shallow_parse [tag_type=NP]>{print}
<pos [tag_type=\)]>
```

3.4 Other Features of LQL

The examples above do not capture all the features of LQL[7], so we describe some of the rest below.

For *spanning* hierarchical layers we can have hierarchical queries with several nested references to the same layer. The following query finds a PP of the form preposition+NP and prints that NP:

```
<full_parse [tag_type=PP]
   ^<pos [tag_type=prep]>
   <full_parse [tag_type=NP]>{print} $>
```

The keyword `noorder` can be used to allow an arbitrary order for the tokens within a layer, e.g.:

```
<sentence [noorder]
   <gene_protein> <pos [tag_type=verb]>
> {print sentence}
```

The language allows for a combination of ordered and unordered constraints. For example, to require a preposition to immediately follow a verb, while allowing a gene/protein name to come before or after this combination, we write:

```
<sentence [noorder]
  <gene_protein>
  (<pos [tag_type=verb] <word [lex=binds]>>
   <pos [tag_type=prep] <word [lex=to]>>)
> {print sentence}
```

Regular expression operators can appear within `()`: e.g. `[lex=(*cellular)]` matches both *intercellular* and *extracellular*. Entire layers are matched with the restriction that a variable number of columns are not considered by the query. For example, a sentence containing many proteins can be expressed as:

```
<sentence [(<gene_protein>+)]>
```

However, this query results in a variable number of columns, which complicates its processing. To disambiguate, a verification during translation should assert the number of columns is fixed. Thus we allow limited regular expressions (wildcards) over the *contents* of an individual token (to be handled by the SQL `LIKE` operator).

Finally, LQL has no overlap operator for retrieval, as it is not clear whether it is needed, but could be accommodated by adding a special character to the open and closing brackets.

3.5 LQL and SQL

LQL can be automatically translated into SQL, although this is not yet implemented[8]. In addition, SQL code can be written to surround the LQL query and reference its results, thus allowing the use of SQL group functions such as `GROUP BY`, `COUNT`, `DISTINCT`, `ORDER`, etc., as well as set operations like `UNION`. An added advantage of the language is that the queries do not need to be modified, if the underlying architecture is changed.

New annotations are added to the database via a Java API: the user needs to specifies document ID, section, layer ID and positional information.

[7] A detailed description can be found online at (url anonymous for reviewing)

[8] LQL can be translated into SQL both outside and inside the database, e.g. by a stored function that constructs the corresponding SQL query, executes it and returns a table.

Arch 1-4	F	*DOCID +SECTION +LAYER_ID +START_CHAR_POS +END_CHAR_POS +TAG_TYPE
Arch 1-4	I	LAYER_ID +TAG_TYPE +DOCID +SECTION +START_CHAR_POS +END_CHAR_POS
Arch 2	F	DOCID +SECTION +LAYER_ID +SEQUENCE_POS +TAG_TYPE +START_CHAR_POS +END_CHAR_POS
Arch 2	I	LAYER_ID +TAG_TYPE +DOCID +SECTION +SEQUENCE_POS +START_CHAR_POS +END_CHAR_POS
Arch 3-4	F	DOCID +SECTION +LAYER_ID +SENTENCE +SEQUENCE_POS +TAG_TYPE +START_CHAR_POS +END_CHAR_POS
Arch 3-4	I	LAYER_ID +TAG_TYPE +DOCID +SECTION +SENTENCE +SEQUENCE_POS +START_CHAR_POS +END_CHAR_POS
Arch 4	I	WORD_ID +LAYER_ID +TAG_TYPE +DOCID +SECTION +START_CHAR_POS +END_CHAR_POS +SENTENCE +SEQUENCE_POS
Arch 5	F	*DOCID +SECTION +LAYER_ID +SENTENCE +FIRST_WORD_POS +LAST_WORD_POS +TAG_TYPE
Arch 5	I	LAYER_ID +TAG_TYPE +DOCID +SECTION +SENTENCE +FIRST_WORD_POS +LAST_WORD_POS
Arch 5	I	WORD_ID +LAYER_ID +TAG_TYPE +DOCID +SECTION +SENTENCE +FIRST_WORD_POS

Table 1: Indexes for the four architectures (F/I = Forward/Inverted and '*' = clustering).

3.6 Discussion

We believe LQL is at least as expressive as TIQL, Emu, and MATE, which lack a clear distinction between sequence ordering and juxtaposition. LQL also supports hierarchies inside the same layer, allows matching across various levels of an *ontological* hierarchy and permits hierarchical descent, a property not considered in Emu and MATE, which support *spanning* only but not hierarchical tokens.

LQL is still a work in progress; this is only the first version of the language and we plan to assess it via usability studies with computational linguistics researchers, modifying it as necessary. However, we feel it is more intuitive and easier to use for text processing than the existing languages.

4 System Architecture

While LQL provides a unified logical view of the annotations, there are many alternatives for the design of the physical representation of these annotations. One can modify the physical representation, and the LQL to SQL translator without modifying existing LQL queries, making LQL a truely declarative language. We present five different physical storage and indexing architectures.

4.1 Record Structure

In all the architectures, the annotations are stored in a relational database. However, the exact record and index structures vary, based on different assumptions about the kinds of annotations and the query workloads. Our basic model is similar to the one of TIPSTER (Grishman, 1996): each annotation is stored as a record, which specifies the character-level beginning and end positions, the layer and the type.

Architecture 1 contains the following columns: **1) docid**: document ID; **2) section**: *title*, *abstract* or *body text*; **3) layer_id** a unique identifier of the annotation layer (*word*, *POS*, *shallow parse sentence* etc.); **4) start_char_pos**: starting character position, relative to particular *section* and *docid*; **5) end_char_pos**: end character position; **6) tag_type**: a layer-specific token unique identifier.

There is a separate table mapping token IDs to entities (the string in case of a word, the MeSH label(s) in case of a MeSH term etc.)

Architecture 2 introduces one additional column, **sequence_pos**, thus defining an ordering for each layer. This simplifies some SQL queries as there is no need for "NOT EXISTS" self joins, which are required under *Architecture 1* in cases where tokens from the same layer must follow each other *immediately*. The *sequence_pos* ordering is well defined for, e.g., the *POS* layer, but not for *full parse*, which is hierarchical. To solve this problem, given a particular *document*, *section* and *layer*, we find and sort the corresponding tokens by *start_char_pos*. Then we set to i the *sequence_pos* for all tokens beginning at s_i ($1 \leq i \leq n$), where $s_1 < s_2 < ... < s_n$ are the ordered unique *start_char_pos* values.

Architecture 3 adds **sentence_id**, which is the number of the current sentence and redefines *sequence_pos* as relative to both *layer_id* and *sentence_id*. This simplifies most queries (removing one join, unless the *sentence* containing the successful match is to be returned), since the patterns are often limited to the same sentence.

Architecture 4 merges the *word* and *POS* layers, and adds **word_id** assuming a one-to-one correspondence between them. This reduces the number of stored annotations and the number of joins in queries with both *word* and *POS* constraints (e.g., as in our first sample query: the word is *correlated* and the POS is *verb*).

Architecture 5 replaces *sequence_pos* with

Queries	(a) 54 303.38 8					(b) 11 77.45 6					(c) 50 1.64 5					(d) 1 16701 4				
Architecture	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
SQL lines	37	37	34	29	29	91	77	75	65	50	45	38	38	39	41	59	50	53	53	35
# Joins	6	6	6	5	5	12	11	11	9	7	7	6	6	6	6	7	7	7	7	4
Time (sec)	3.98	4.35	3.59	1.69	1.94	3.88	5.68	5.41	3.85	3.55	17.90	23.42	21.49	30.07	4.06	1879	1700	2182	1682	1582

Table 2: Statistics per workload and architecture. For each query type (a-d) are shown averages for number of queries/type, number of results/query, and number of lines of LQL/query.

first_word_pos and **last_word_pos**, which correspond to the *sequence_pos* of the first/last *word* covered by the annotation. This representation treats the *word* layer as atomic and requires all annotation boundaries to coincide with word boundaries. In cases where this assumptions does not hold, a more fine-grained level can be used instead (e.g. characters without white-spaces). This architecture alleviates the main limitation of the previous ones by coping naturally with adjacency constraints between different layers and allows for a simpler indexing structure.

4.2 Indexing Structure

The benefits of a relational database cannot be realized without a good indexing structure. Indexes are essential when scaling to large number of documents and annotations. We use two types of *composite* indexes: *forward* and *inverted*. An index lookup can be performed on any column combination that corresponds to an index *prefix*. The *forward* indexes support lookup based on position in a given document, while the *inverted* ones support lookup based on annotation *values* (i.e., *tag_type* and *word_id*)[9]. An RDBMS's query optimizer estimates the optimal access paths (index and table scans), and join orders based on statistics collected over the stored records. In complex queries a combination of *forward* and *inverted* indexes is typically used. Table 1 lists the indexes used in each architecture. The records are clustered based on their primary key (marked as '*'), ensuring that locally related annotations are stored physically close together to reduce physical disk I/O's.

5 Evaluation

We annotated 13,504 MEDLINE abstracts using the Stanford Lexicalized Parser (Klein and Manning, 2003) for sentence splitting, word tokenization, POS tagging and parsing. We wrote a shallow parser and tools for gene and MeSH term recognition. This resulted in 10,910,243 records (about 3 times the number of words) stored in an IBM DB2 Universal Database Server[10]. Table 3 summarizes the storage requirements.

We defined workloads based on variants of the queries (a-d) from Section 3. We ran the experiments on a dual processor Dell Precision 450 with 2GB of RAM (only 1GB was allocated to the database), running Red Hat Linux 8.0. Table 2 summarizes the results. It is clear that different architectures are optimized for different types of queries. However, *Architecture 5* performs well (if not best) on all query types, while the other architectures perform poorly on at least one query type. Since the storage requirement of *Architecture 5* is comparable to that of *Architecture 1* and results in much simpler queries, we recommend it as the best architecture for storing text annotations. The main limitation of *Architecture 5* is the requirement that an atomic annotation layer be defined. If this is not the case, *Architecture 1* seems to provide the best alternative.

To evaluate the scalability of *Architecture 5*, we ran a combined workload, with a random set from the first 3 query types, with varying database buffer pool sizes (memory). The corresponding buffer pool sizes, elapsed-times (sec), and buffer read times (sec) are: 1GB/2.3/1.05, 100MB/2.9/1.67, 10MB/4.6/3.34, 1MB/8.3/6.25. These results suggest that the query execution time grows as a sub-linear (log) function of the memory size. We believe a similar ratio will be observed when increasing database size and keeping the memory size fixed, (e.g., a database 1000 times bigger will run only about 6 times slower),

[9] Our *inverted* indexes can be seen as a direct extension of the widely used *inverted file* indexes in traditional IR systems.

[10] http://www-306.ibm.com/software/data/db2/udb/

	Architecture				
Space (MB)	**1**	**2**	**3**	**4**	**5**
Data Storage	168.5	168.5	168.5	132.5	136.5
Index Storage	617	1397	1441	1182	673.5

Table 3: Architecture storage requirements.

thus making it feasible to query the whole of MEDLINE ($\sim$12 million abstracts) within minutes. Further, the annotations can easily be partitioned based on *document_id*, enabling parallel query executions on multiple machines.

6 Conclusions and Future Work

We have provided a mechanism to effectively store and query layers of textual annotations. Using a collection of 13,504 MEDLINE abstracts, we have evaluated various structures for data storage and have arrived at an efficient and simple one. We used variations of queries drawn from published research, to ensure the real-world applicability of the experiments. Although illustrated using semantic markup from the biosciences, users are free to use any kind of ontology to create the layers.

We also presented a concise language (LQL) to express queries that span multiple levels of the annotation structure, which captures the user's intent better as the syntax is more intuitive and closely resembles the annotation structure. We plan to conduct a usability study to assess this claim.

Future work includes automating the LQL to SQL translation process and testing the scalability of this approach on larger document collections.

Acknowledgements This research was supported by a grant from the ARDA AQUAINT program, NSF DBI-0317510, and a gift from Genentech.

References

Steven Bird, Peter Buneman, and Wang-Chiew Tan. 2000. Towards a Query Language for Annotation Graphs. *Second International Conference on Language Resources and Evaluation*, 807–814.

Steven Bird and Jonathan Harrington. 2001. Speech annotation and corpus tools. *Speech Communication*, 33(1-2):1-4.

Steven Bird and Mark Liberman. 2001. A formal framework for linguistic annotation. *Speech Communication*, 33(1–2):23–60.

Christian Blaschke, Miguel A. Andrade, Christos Ouzounis, Alfonso Valencia. 1999. Automatic Extraction of Biological Information From Scientific Text: Protein-Protein Interactions. *International Conference on Intelligent Systems for Molecular Biology:ISMB*, 60–67.

Steve Cassidy. 1999. Compiling Multi-tiered Speech Databases into the Relational Model: Experiments with the Emu System. *6th European Conference on Speech Communication and Technology Eurospeech 99*, 2127–2130, Budapest, Hungary.

Steve Cassidy and Steven Bird. 2000. Querying Databases of Annotated Speech. *Database Technologies: Proceedings of the Eleventh Australasian Database Conference, IEEE Computer Society*, 12–20.

Steve Cassidy and Jonathan Harrington. 2001. Speech annotation and corpus tools. *Speech Communication*, 33(1–2):61–77.

Ralph Grishman. 1996. Building an Architecture: a CAWG Saga. *Advances in Text Processing: Tipster Program Phase II*, Morgan Kaufmann, 1996.

Dan Klein and Christopher D. Manning. 2003. Fast Exact Inference with a Factored Model for Natural Language Parsing. *Advances in Neural Information Processing Systems 15, MIT Press*, 3–10.

Xiaoyi Ma, Haejoong Lee, Steven Bird and Kazuaki Maeda. 2002. Models and Tools for Collaborative Annotation. *Third International Conference on Language Resources and Evaluation*, 2066–2073.

David McKelvie, Amy Isard, Andreas Mengel, Morten B. Møller, Michael Grosse and Marion Klein. 2001. Speech annotation and corpus tools. *Speech Communication*, 33(1–2):97–112.

Andreas Mengel, Ulrich Heid, Arne Fitschen, and Stefan Evert. 1999. Specification of Coding Workbench: Improved Query Language (q4m). *Technical Report MATE Deliverable D3.1, Stuttgart: Institut fur Maschinelle Sprachverarbeitung*.

Goran Nenadic, Hideki Mima, Irena Spasic, Sophia Ananiadou and Jun-ichi Tsujii. 2002. Terminology-Driven Literature Mining and Knowledge Acquisition in Biomedicine. *International Journal of Medical Informatics*, 67:33–48.

James Pustejovsky, José Castaño, Brent Cochran, Maciej Kotecki and Michael Morrell. 2001. Automatic Extraction of Acronym-Meaning Pairs from MEDLINE Databases. *Medinfo*, 10(Pt 1)371–385.

Barbara Rosario, Marti A. Hearst and Charles Fillmore. 2002. The Descent of Hierarchy, and Selection in Relational Semantics. *ACL*, 247–254.

James Thomas, David Milward, Christos Ouzounis, Stephen Pulman and Mark Carroll. 2000. Automatic Extraction of Protein Interactions from Scientific Abstracts. *Pacific Symposium on Biocomputing*, 541–551.