Principles, Implementation Strategies, and Evaluation of a Corpus Query System

Ulrik Petersen

University of Aalborg
Department of Communication and Psychology
Kroghstræde 3, DK — 9220 Aalborg East, Denmark
ulrikp@hum.aau.dk
http://emdros.org/

Abstract. The last decade has seen an increase in the number of available corpus query systems. These systems generally implement a query language as well as a database model. We report on one such corpus query system, and evaluate its query language against a range of queries and criteria quoted from the literature. We show some important principles of the design of the query language, and argue for the strategy of separating what is retrieved by a linguistic query from the data retrieved in order to display or otherwise process the results, stating the needs for generality, simplicity, and modularity as reasons to prefer this strategy.

1 Introduction

The last decade has seen a growth in the number of available corpus query systems. Newcomers since the mid-1990ies include MATE Q4M [1], the Emu query language [2], the Annotation Graph query language [3], TIGERSearch [4], NXT Search [5], TGrep2 [6], and LPath [7].

Our own corpus query system, Emdros [8, 9], has been in development since 1999. It is based on ideas from the PhD thesis by Crist-Jan Doedens [10]. It implements a database model and a query language which are very general in their applicability: Our system can be applied to almost any linguistic theory, almost any linguistic domain (e.g., syntax, phonology, discourse) and almost any method of linguistic tagging. Thus our system can be used as a basis for implementing a variety of linguistic applications. We have implemented a number of linguistic applications such as a generic query tool, a HAL[1] space, and a number of import tools for existing corpus formats. As the system is Open Source, others are free to implement applications for their linguistic problem domains using our system, just as we plan to continue to extend the range of available applications.

The rest of the paper is laid out as follows: First, we briefly describe the EMdF database model underlying Emdros, and give an example of a database expressed in EMdF. Second, we describe the MQL query language of Emdros and its principles. Third, we argue for the strategy of separating the process of

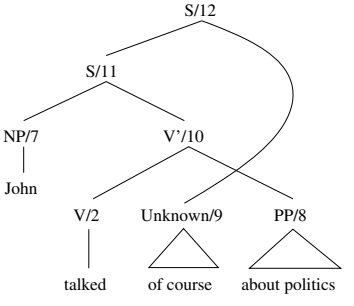
[1] HAL here stands for "Hyperspace Analogue to Language," and is a statistical method based on lexical co-occurrence invented by Dr. Curt Burgess and his colleagues [11].

A. Yli-Jyrä, L. Karttunen, and J. Karhumäki (Eds.): FSMNLP 2005, LNAI 4002, pp. 215–226, 2006.

retrieving linguistic query results from the process of retrieving linguistic objects based on such results for application-specific purposes. Fourth, we evaluate MQL against a set of standard queries and criteria for corpus query languages culled from the literature. Finally, we conclude the paper.

2 The EMdF Database Model

To illustrate how data can be stored in Emdros, consider Fig. 1. It shows an example of a discontiguous clause, taken from [12, p. 95], represented both as a tree and as a database expressed in the EMdF database model.



a. A tree with a discontiguous clause, adapted from [12, p. 95]

	1	2	3	4	5	6
Word	id: 1 surf.: John pos: NProp parent: 7	id: 2 surf.: talked pos: V parent: 10	id: 3 surf.: of pos: P parent: 9	id: 4 surf.: course pos: N parent: 9	id: 5 surf.: about pos: P parent: 8	id: 6 surf.: politics pos: N parent: 8
Phrase	id: 7 type: NP parent: 11		id: 9 type: Unknown parent: 12		id: 8 type: PP parent: 10	
Phrase		id: 10 type: V' parent: 11			id: 10 type: V' parent: 11	
Clause	id: 11 type=S parent: 12				id: 11 type=S parent: 12	
Clause	id: 12 type=S					

b. A EMdF representation of the tree

Fig. 1. Two representation of a tree with a discontiguous clause

At the top of Fig. 1.b. are the *monads*. A monad is simply an integer, and the sequence of the monads defines the logical reading order. An object is a (possibly discontiguous) set of monads belonging to an object type (such as "Word", "Phrase", "Clause"), and having a set of associated attribute-values. The object type of an object determines what attributes it has. For example, the "Word" object type in Fig. 1.b has attributes "id", "surface", "pos" (part of speech), and "parent". The id is a database-widely unique number that identifies that object. In the above database, this has been used by the "parent" attribute to point to the immediately dominating node in the tree.

In the EMdF database model, object attributes are strongly typed. The model supports strings, integers, ids, and enumerations as types for attributes, as well as lists of integers, ids, and enumeration labels. Enumerations are simply sets of labels, and have been used for the Word.pos, Phrase.type, and Clause.type attributes in the figure.[2] Real-number values are under implementation, and will be useful for, e.g., acoustic-signal timelines.

[2] The "dot-notation" used here is well known to programmers, and is basically a possessive: "Word.pos" means "the pos attribute **of** the Word object-type".

3 The MQL Query Language

The MQL query language of Emdros is a descendant of the QL query language described in [10]. Like QL, it is centered around the concept of "blocks", of which there are three kinds: "Object blocks", "gap blocks", and "power blocks".

An "Object block" finds objects in the database (such as phonemes, words, phrases, clauses, paragraphs, etc.) and is enclosed in [square brackets]. For example, the query `[Word surface="saw"]` will find `Word` objects whose `surface` attribute is "saw", whereas the query `[Phrase type = NP and function = Subj]` will find phrases whose phrase type is NP and whose function is Subject. Of course, this presupposes an appropriately tagged database. The attribute-restrictions on the object are arbitrary boolean expressions providing the primitives "AND", "OR", "NOT", and "grouping (parentheses)". A range of comparison-operators are also provided, including equality, inequality, greater-than (or equal to), less than (or equal to), regular expressions (optionally negated), and `IN` a disjoined list of values. For lists, the `HAS` operator looks for a specific value in the list.

A "gap block" finds "gaps" in a certain context, and can be used to look for (or ignore) things like embedded relative clauses, postpositive conjunctions, and other material which is not part of the surrounding element. A gap block is specified as `[gap ...]` when obligatory, and as `[gap? ...]` when optional.

A "power block" is denoted by two dots ("`..`"), and signifies that there can be arbitrary space between the two surrounding blocks. However, this is always confined to be within the limits of any context block.

The power block can optionally have a restriction such as "`.. <= 5`" or "`.. BETWEEN 3 AND 6`" meaning respectively that the "space" can be between zero and five "least units" long, or that it must be between 3 and 6 "least units" long. Precisely what the "least unit" is, is database-dependent, but is usually "Word" or "Phoneme".[3]

The MQL query language implements the important principle of *topographicity* described in [10], meaning that there is an isomorphism between the structure of the query and the structure of the objects found. The principle of topographicity works with respect to two important textual principles, namely embedding and sequence.

As an example of topographicity with respect to embedding, consider the query Q1 in Fig. 3 on page 221. This query finds sentences within which there is at least one word whose surface is "saw". The "`[Word surface="saw"]`" object block is *embedded in* the "`[Sentence ...]`" object block. Because of the principle of topographicity, any Word objects found must also be *embedded in* the Sentence objects found.

Similarly, in Query Q5 in Fig. 3, the two inner `[Syntax level=Phrase ...]` object blocks find Syntax objects that immediately follow each other in sequential order, because the object blocks are adjacent. "Being adjacent" here means

[3] This is an example of the generality of the EMdF database model, in that it supports many different linguistic paradigms and methods of analysis.

"not being separated by other blocks" (including a power block). There is a caveat, however. The default behavior is to treat objects in the database as "being adjacent" even if they are separated by a gap in the surrounding context. For examle, in Query Q5, if the surrounding Sentence object has a gap between the NP and the VP[4], then that query will find such a sentence due to the default behavior. If this is not the desired behavior (i.e., gaps are not allowed), one can put the "`!`" (bang) operator in between the object blocks, as in Query Q4 in Fig. 3. This will require the objects found by the object blocks surrounding the bang to be strictly sequential.

An object block can be given the restriction that it must be `first`, `last`, or `first and last` in its surrounding context. An example using the `last` keyword can be seen in Query Q3 in Fig. 3.

The object retrieved by an object block can be given a name with the `AS` keyword. Subsequent object blocks can then refer back to the named object. An example can be seen in Query Q5 in Fig. 3, where the dominating Syntax object is named `AS S1`. The dominated phrase-level Syntax object blocks then refer back to the dominating object by means of the "possessive dot notation" mentioned previously. Obviously, this facility can be used to specify both agreement, (immediate) dominance, and other inter-object relationships.

The NOTEXIST operator operates on an object block to specify that it must not exist in a given context. An example can be seen in Query Q2 in Fig. 3, where the existence of a word with the surface "saw" is negated. That is, the query finds sentences in which the word "saw" does not occur.

Notice that this is different from finding sentences with words whose *surface* is not "saw", as the query `[Sentence [Word surface<>"saw"]]` would find. Relating this to First Order Logic, the NOTEXIST operator is a negated existential quantifier $\neg\exists$ at *object* level, whereas the `<>` operator is a negated equality operator $\neq$ at object *attribute* level. If the NOTEXIST operator is applied to an object block, the object block must be the only block in its context.

The Kleene Star operator also operates on an object block, and has the usual meaning of repeating the object block zero or more times, always restricted to being within the boundaries of any surrounding context block. For example, the query

```
[Sentence
   [Word pos=preposition]
   [Word pos IN (article,noun,adjective,conjunction)]*
]
```

would find the words of many prepositional phrases, and could be used in a stage of initial syntactic markup of a corpus. The Kleene Star also supports restricting the number of repetitions with an arbitrary set of integers. For example: `[Phrase]*{0,1}` means that the Phrase object may be repeated 0 or 1

[4] As argued by [12], the sentence "John, of course, talked about politics" is an example of an element with a gap, since "of course" is not part of the surrounding clause.

times;[5] `[Clause]*{2-4}` means that the Clause object may be repeated 2, 3, or 4 times; and any set of integers can be used, even discontiguous ones, such as `[Phrase]*{0-3,7-9,20-}`. The notation "`20-`" signifies "from 20 to infinity".

An `OR` operator operating on strings of blocks is available. It means that one or both strings may occur in a given context. An example is given in Query Q7 in Fig. 3.

MQL has some shortcomings, some of which will be detailed later. Here we will just mention four shortcomings which we are working to fix, but which time has not allowed us to fix yet. We have worked out an operational semantics for the following four constructs: AND between strings of blocks (meaning that both strings must occur, and that they must overlap);[6] Grouping of strings of blocks; and general Kleene Star on strings of blocks (the current Kleene Star is only applicable to one object block). A fourth operator can easily be derived from the existing OR construct on strings of blocks, namely permutations of objects.

4 Retrieval of Results

When querying linguistic data, there are often three distinct kinds of results involved:

1. The "**meat**", or the particular linguistic construction of interest.
2. The **context**, which is not exactly what the user is interested in, but helps delimit, restrict, or facilitate the search in some way. For example, the user may be interested in subject inversion or agentless passives, but both require the context of a sentence. Similarly, the user may be interested in objects expressed by relative pronouns combined with a repeated pronoun in the next clause, which might require the presence of intervening, specified, but otherwise non-interesting material such as a complementizer.[7] In both cases, the user is interested in a specific construction, but a certain context (either surrounding or intervening) needs to be present. The context is thus necessary for the query to return the desired results, but is otherwise not a part of the desired results.
3. The **postprocessing** results which are necessary for purposes which are outside the scope of the search.

To illustrate, consider the query Q2 in Fig. 3. For display purposes, what should be retrieved for this query? The answer depends, among other things, on the linguistic domain under consideration (syntax, phonology, etc.), the linguistic categories stored in the database, the purposes for which the display is made, and the sophistication of the user. For the domain of syntax, trees might

[5] Notice that this supports optionality in the language; that the phrase object appears 0 or 1 times is equivalent to saying that it is optional.

[6] This is precisely what is needed for querying overlapping structures such as those found in speech data with more than one speaker, where the speaker turns overlap.

[7] E.g., "He gave me a ring, which, I really don't like that it is emerald."

be appropriate, which would require retrieval of all nodes dominated by the sentence. For the domain of phonology, intonational phrases, tones, pauses, etc. as well as the phonemes dominated by the sentence would probably have to be retrieved. As to purpose, if the user only needed a concordance, then only the words dominated by the sentence need be retrieved, whereas for purposes requiring a full-fledged tree, more elements would have to be retrieved. The level of sophistication of the user also has a role to play, since an untrained user might balk at trees, whereas keywords in context may be more understandable.

Similarly, for statistical purposes, it is often important to retrieve frequency counts over the entire corpus to compare against the current result set. These frequency counts have nothing to do with the answer to the original query, but instead are only needed after the results have been retrieved. They are, in a very real sense, outside the scope of the query itself: The user is looking for a particular linguistic construction, and the corpus query system should find those constructions. That the post-query purpose of running the query is statistical calculations is outside the scope of the query, and is very application-specific.

Thus what is asked for in a linguistic query is often very different from what needs to be retrieved eventually, given differences in linguistic domain, categories in the database, purpose of display, and sophistication of the user. Therefore, in our view, it is advantageous to split the two operations into separate query language constructs. The subset of the query language supporting linguistic querying would thus be concerned with returning results based on what is asked for in a linguistic query, whereas other subsets of the query language would be concerned with retrieving objects based on those results for display- or other purposes.

This separation, because it is general, supports a multiplicity of linguistic applications, since the concern of linguistic querying (which is common to all linguistic query applications) is separated from the concern of querying for display-, statistical, or other purposes (which are specific to a given application). Moreover, it shifts the burden of what to retrieve based on a given query (other than what is being asked for) off the user's mind, and onto the application, thus making the query language simpler both for the user and for the corpus query system implementor. Finally, this strategy lends itself well to modularization of the query language. That modularization is good, even necessary for correct software implementation has long been a credo of software engineering.[8]

5 Evaluation

Lai and Bird [13] formulate some requirements for query languages for treebanks. They do so on the backdrop of a survey of a number of query languages, including TGrep2, TIGERSearch, the Emu query language, CorpusSearch, NXT Search, and LPath. Lai and Bird set up a number of test queries (see Fig. 2) which are then expressed (or attempted expressed) in each of the surveyed query languages.

[8] Emdros adheres to this modular principle of separation of concerns between corpus query system and a particular linguistic application on top of it.

Q1. Find sentences that include the word 'saw'.
Q2. Find sentences that do not include the word 'saw'.
Q3. Find noun phrases whose rightmost child is a noun.
Q4. Find verb phrases that contain a verb immediately followed by a noun phrase that is immediately followed by a prepositional phrase.
Q5. Find the first common ancestor of sequences of a noun phrase followed by a verb phrase.
Q6. Find a noun phrase which dominates a word *dark* that is dominated by an intermediate phrase that bears an L-tone.
Q7. Find a noun phrase dominated by a verb phrase. Return the subtree dominated by that noun phrase.

Fig. 2. The test queries from [13], Fig. 1

```
Q1. [Sentence
       [Word surface="saw"]
    ]
Q2. [Sentence
       NOTEXIST [Word
             surface="saw"]
    ]
Q3. [Phrase type=NP
       [Word last pos=noun]
    ]
Q4. [Phrase type=VP
       [Word pos=verb]!
       [Phrase type=NP]!
       [Phrase type=PP]
    ]
```

```
Q5.? [Syntax AS S1
        [Syntax level=Phrase AND type=NP
           AND parent=S1.id]
        [Syntax level=Phrase AND type=VP
           AND parent=S1.id]
     ]
Q6.? [Intermediate tone="L-"
        [Phrase type=NP
           [Word surface="dark"]
        ]
     ]
Q7.  [Phrase type=VP
        [Phrase type=NP AS np1
           [Phrase parents HAS np1.id
              [Word]
           ] OR
           [Word parent=np1.id]
        ]
     ]
```

Fig. 3. MQL queries for Q1-Q7

For all query languages surveyed, it is the case that at least one query cannot be correctly expressed.

The queries are attempted expressed in MQL as in Fig. 3. Query Q1 is trivial, and performs as expected. Query Q2 has already been explained above, and deserves no further comment. The constraint of query Q3 that the noun must be the rightmost child is elegantly expressed by the "`last`" operator on the noun.

In query Q4, the verb, the NP, and the PP are not separated by power blocks ("`..`") and so must immediately follow each other. As mentioned above, gaps are ignored unless the "bang" operator ("`!`") is applied in between the object blocks. Since the query specification explicitly mentions "immediately followed by", we have chosen to insert this operator. Of course, if the default behavior is desired, the bang operator can simply be left out.

Query Q5 fails to yield the correct results in some cases because it presupposes that the "first common ancestor" is the immediate parent, which it need not be. Had the "`parent=S1.id`" terms been left out of the conjunctions, the query would have found all ancestors, not just the immediate ancestor. It is a shortcoming of the current MQL that it is not easy to express other relationships than "general ancestry" and "immediate ancestry".

Query Q5 also presupposes a different database structure than the other queries: In the database behind Q5, all syntax-level objects have been lumped together into one "Syntax" type. This "Syntax" type has a "level" attribute specifying the linguistic level at which the element occurs (Phrase, Clause, etc.), as well as other attributes.

This reorganization of the database is necessary for Q5 because it does not specify what level the dominating node should be at (Phrase, Clause, or Sentence). It is a limitation in Emdros that it can only handle one, explicit type for each object block.

For some lingusitic databases, query Q6 would fail to retrieve all possible instances because it assumes that the NP is wholly contained in the Intermediate Phrase. But as [14, p. 176] reports, this is not always true.[9]

Query Q7 not only needs to specify context, but also to retrieve the subtree, presumably for display- or other purposes, since it is not part of what is being asked for (i.e., the "meat"). As mentioned in Sect. 4, Emdros adheres to a different philosophy of implementation. While it is possible in MQL to retrieve exactly whatever the user wants, the algorithm for doing so would in most cases be split between retrieving linguistic results and using other parts of the query language for retrieving objects for display-purposes.

The Q7 query nevertheless fulfills its purpose by retrieving all phrases dominated by the NP together with the words they contain, `OR` all words immediately dominated by the NP. Thus, Emdros is able to fulfill the purpose of the query even though Emdros was not designed for such use.

Lai and Bird go on from their survey to listing a number of requirements on linguistic query languages. The first requirement listed is "accurate specification of the query tree". Lai and Bird give eight subtree-matching queries, all of which can be expressed in MQL (see Fig. 4). Query number 5 would require the employment of the technique used for query Q5 in Fig. 3 of using a single object type for all syntax objects, using an attribute for the syntactic level, then leaving out the level from the query.

Another requirement specified by Lai and Bird is that of reverse navigation, i.e., the need to specify context in any direction. MQL handles this gracefully, in our opinion, by the principle of topographicity with respect to embedding and sequence. Using this principle, any context can be specified in both vertical directions, as well as along the horizontal axis.

[9] The example given there is an intermediate phrase boundary between adjectives and nouns in Japanese — presumably the adjective and the noun belong in the same NP, yet the intermediate phrase-boundary occurs in the middle of the NP.

1. Immediate dominance: A dominates B, A may dominate other nodes.	`[A AS a1 [B parent=A1.id]]`
2. Positional constraint: A dominates B, and B is the first (last) child of A.	`[A [B first] ]` or: `[A [B last] ]`
3. Positional constraint with respect to a label: A dominates B, and B is the last B child of A.	`[A [B last] ]`
4. Multiple Dominance: A dominates both B and C, but the order of B and C is unspecified.	`[A [B]..[C] OR [C]..[B]]`
5. Sibling precedence: A dominates both B and C, B precedes C; A dominates both B and C, B immediately precedes C, and C is unspecified.	precedes: `[A [B]..[C]]` immediately precedes: `[A [B][C]]` or `[A [B]![C]]`.

6. Complete description: A dominates B and C, in that order, and nothing else.	`[A as a1`
	`  [B first parent=a1.id]!`
	`  [B last parent=a1.id]`
	`]`

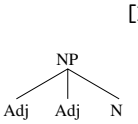
7. Multiple copies: A dominates B and B, and the two Bs are different instances.	`[A [B]..[B]]`
8. Negation: A does not dominate node with label B.	`[A NOTEXIST [B] ]`

Fig. 4. Subtree queries in the MQL query language, after Lai and Bird's Fig. 9

Lai and Bird then mention non-tree navigation as a requirement. They give the example of an NP being specified either as "[NP Adj Adj N]" or as "[NP Adj [NP Adj N]]", the latter with a Chomsky-adjoined NP inside the larger NP. MQL handles querying both structures with ease, as seen in Fig. 5. Note that the query in Fig. 5.a. would also find the tree in Fig. 5.b. Thus non-tree navigation is well supported.

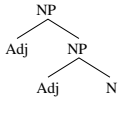
Furthermore, Lai and Bird mention specification of precedence and immediate precedence as a requirement. MQL handles both with ease because of the principle of topographicity of sequence. General precedence is signified by the power block ("`..`"), whereas immediate precedence is signified by the absence of the power block, optionally with the bang operator ("`!`").

Lai and Bird then discuss closures of various kinds. MQL is closed both under dominance (by means of topographicity of embedding) and under precedence




```
[Phrase type=NP
  [Word first
       pos=adjective]
  [Word pos=adjective]
  [Word last pos=noun]
]
```

a. Flat structure




```
[Phrase type=NP
  [Word first pos=adjective]
  [Phrase last type=NP
    [Word first pos=adjective]
    [Word last pos=noun]
  ]
]
```

b. Chomsky-adjoined structure

Fig. 5. Queries on NP structure

and sibling precedence (by means of topographicity of sequence, as well as the power block and the `AS` keyword, which separately or in combination can be used to specify closures under both relationships). MQL is also closed under atomic queries involving one object (by means of the Kleene Star).[10]

Lai and Bird discuss the need for querying above sentence-level. Since the EMdF database model is abstract and general, the option exists of using ordered forests as mentioned by Lai and Bird. The MQL query language was designed to complement the EMdF model in its generality, and thus querying over ordered forests is well supported using the principle of topographicity of sequence combined with the `AS` construct. Thus the MQL language is not restricted to querying sentence-trees alone, but supports querying above sentence-level.

Another requirement mentioned by Lai and Bird is that of integration of several types of lingusitic data, in particular using intersecting hierarchies and lookup of data from other sources. The EMdF model supports intersecting hierarchies well. MQL, however, because of the principle of topographicity of embedding and the lack of an AND construct between strings of blocks, does not currently support querying of intersecting hierarchies very well, as illustrated by the failure of Query Q6 in Fig. 3 to be correct. Thus Emdros currently falls short on this account, though an AND construct is planned.

There is also currently a lack of support for querying data from other sources. However, this can be implemented by the application using Emdros, provided the data from other sources can be known before query-time and can thus be written into the query. This would, of course, presuppose that the application does some kind of rewriting of the query made by the user.

The final requirement mentioned by [13] is the need to query non-tree structure. For example, the TIGER Corpus [15] includes secondary, crossing edges, and the Penn Treebank includes edges for WH-movement and topicalization [16]. MQL handles querying these constructions by means of the `AS` keyword and referencing the ID of the thus named object, as in Query Q5 in Fig. 3.

6 Conclusion and Further Work

We have presented the EMdF database model and the MQL query language of our corpus query system, Emdros. We have shown how the data to be retrieved for display-, statistical, or other purposes can often be different from what is asked for in a linguistic query, differentiating between "meat", "context", and "postprocessing results". On the basis of this distinction, we have argued for the strategy of separating the process of lingusitic querying from the process of retrieval of data for display- or other purposes. This implementation strategy of separation of concerns gives rise to the benefits of generality of the language (and thus its applicability to a wide variety of linguistic applications), simplicity of the language (and thus ease of use for the user), and modularity (and thus ease

[10] Once we have implemented the general Kleene Star on strings of blocks, MQL will be closed under atomic queries involving more than one block.

of implementation, maintainability, and attainment of the goal of correctness for the system implementor). Finally, we have evaluated MQL against the queries and requirements of [13], and have shown MQL to be able to express most of the queries, and to meet most of the requirements that [13] puts forth.

However, Emdros falls short on a number of grounds. First, although its database model is able to handle intersecting hierarchies, its query language does not currently handle querying these intersecting hierarchies very well. This can be fixed by the inclusion of an AND operator between strings of object blocks. Second, a general Kleene Star is lacking that can operate on groups of (optionally embedded) objects. Third, the query language currently only supports one, explicit object type for any given object block. This can be fixed, e.g., by introducing true object orientation with inheritance between object types. Fourth, the system currently does not support real numbers as values of attributes of objects, which would be very useful for phonological databases. Fifth, it is currently not easy to express other, more specific dominance relationships than immediate dominance and general dominance. As has been described above, the removal of most of these shortcomings is planned.

Thus Emdros is able to meet most of the requirements being placed on today's linguistic query systems. We have not here fully explored its applicability to phonological or discourse-level databases, since [13] concentrated on treebanks, but that is a topic for a future paper.

References

1. Mengel, A.: MATE deliverable D3.1 – specification of coding workbench: 3.8 improved query language (Q4M). Technical report, Institut für Maschinelle Sprachverarbeitung, Stuttgart, 18. November (1999)
2. Cassidy, S., Bird, S.: Querying databases of annotated speech. In Orlowska, M., ed.: Database Technologies: Proceedings of the Eleventh Australasian Database Conference, volume 22 of Australian Computer Science Communications, Canberra, Australia. IEEE Computer Society (2000) 12–20
3. Bird, S., Buneman, P., Tan, W.C.: Towards a query language for annotation graphs. In: Proceedings of the Second International Conference on Language Resources and Evaluation. European Language Resources Association, Paris (2000) 807–814
4. Lezius, W.: TIGERSearch – ein Suchwerkzeug für Baumbanken. In Busemann, S., ed.: Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002), Saarbrücken. (2002) 107–114
5. Heid, U., Voormann, H., Milde, J.T., Gut, U., Erk, K., Pado, S.: Querying both time-aligned and hierarchical corpora with NXT Search. In: Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May 2004. (2004)
6. Rohde, D.L.T.: TGrep2 user manual, version 1.12. Available for download online http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf. Access Online April 2005 (2004)
7. Bird, S., Chen, Y., Davidson, S., Lee, H., Zheng, Y.: Extending XPath to support linguistic queries. In: Proceedings of Programming Language Technologies for XML (PLANX) Long Beach, California. January 2005. (2005) 35–46

8. Petersen, U.: Emdros — A text database engine for analyzed or annotated text. In: Proceedings of COLING 2004, 20th International Conference on Computational Linguistics, August 23rd to 27th, 2004, Geneva, International Commitee on Computational Linguistics (2004) 1190–1193 http://emdros.org/petersen-emdros-COLING-2004.pdf.
9. Petersen, U.: Evaluating corpus query systems on functionality and speed: Tigersearch and emdros. In Angelova, G., Bontcheva, K., Mitkov, R., Nicolov, N., Nikolov, N., eds.: International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005, Shoumen, Bulgaria, INCOMA Ltd. (2005) 387–391 ISBN 954-91743-3-6.
10. Doedens, C.J.: Text Databases: One Database Model and Several Retrieval Languages. Number 14 in Language and Computers. Editions Rodopi, Amsterdam and Atlanta, GA. (1994)
11. Lund, K., Burgess, C.: Producing high-dimensional semantic spaces from lexical co-occurrence. Behavior Research Methods, Instruments and Computers **28** (1996) 203–208
12. McCawley, J.D.: Parentheticals and discontinuous constituent structure. Linguistic Inquiry **13** (1982) 91–106
13. Lai, C., Bird, S.: Querying and updating treebanks: A critical survey and requirements analysis. In: Proceedings of the Australasian Language Technology Workshop, December 2004. (2004) 139–146
14. Beckman, M.E., Pierrehumbert, J.B.: Japanese prosodic phrasing and intonation synthesis. In: Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics. ACL (1986) 173–180
15. Brants, S., Hansen, S.: Developments in the TIGER annotation scheme and their realization in the corpus I. In: Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002. (2002) 1643–1649
16. Taylor, A., Marcus, M., Santorini, B.: The Penn treebank: An overview. In Abeillé, A., ed.: Treebanks — Building and Using Parsed Corpora. Volume 20 of Text, Speech and Language Technology. Kluwer Academic Publishers, Dordrecht, Boston, London (2003) 5–22