Towards A Query Language for Annotation Graphs

Steven Bird*, **Peter Buneman**† **and Wang-Chiew Tan**†

*Linguistic Data Consortium, University of Pennsylvania, 3615 Market Street, Philadelphia, PA 19104, USA
†Department of Computer Science, University of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104, USA

Abstract

The multidimensional, heterogeneous, and temporal nature of speech databases raises interesting challenges for representation and query. Recently, annotation graphs have been proposed as a general-purpose representational framework for speech databases. Typical queries on annotation graphs require path expressions similar to those used in semistructured query languages. However, the underlying model is rather different from the customary graph models for semistructured data: the graph is acyclic and unrooted, and both temporal and inclusion relationships are important. We develop a query language and describe optimization techniques for an underlying relational representation.

1. Introduction

In recent years, annotated speech databases have grown tremendously in size and complexity. In order to maintain or access the data, one invariably has to write special purpose programs. With the introduction of a general purpose data model, the annotation graph (Bird and Liberman, 1999), it is possible to abstract away from idiosyncrasies of physical format. However, this does not magically solve the maintenance and access problems. In this paper, we contend that some form of query language is essential for annotation graphs, and we report our research on such a language.

Query languages for databases have two, sometimes conflicting, purposes. First they should express – as naturally as possible – a large number of data extraction and restructuring tasks. Second, they should be optimizable. This means that they should be based on a few efficiently implemented primitives; they should also make it easy to discover optimization strategies that may involve query rewriting, execution planning and indexing. The relational algebra and its practical embodiment, SQL, are examples of such languages, however they are unsuitable for annotation graphs first because it is difficult (or impossible – depending on the version of SQL) to express many practical queries, and second because the optimizations that are necessary for annotation graph queries are not in the repertoire of standard relational query optimizations.

The recent development of query languages for semistructured data (Buneman et al., 1996; Quass et al., 1995; Deutsch et al., 1998) offer more natural forms of expression for annotation graphs. In particular, these languages support *regular path patterns* – regular expressions on the labels in the graph – to control the matching of variables in the query to vertices or edges in the graph. While regular path patterns are useful, the usual model of semistructured data, that of a labeled tree, is not appropriate for annotation graphs. In particular, it fails to capture the quasi-linear structure of these graphs, which is essential in query optimization.

After reviewing some existing languages for linguistic annotations, we present the annotation graph model, its relational representation, and some relational queries on annotation graphs. Then we develop a new query language for annotation graphs that allows complex pattern matching. It is loosely based on semistructured query languages, but the syntax simplifies the problem of finding regions of the data that bound the search. Finally, we describe an optimization method that exploits the quasi-linear structure of annotation graphs.

2. Query Languages for Annotated Speech

If linguistic annotations could be modeled as simple hierarchies, then existing query languages for structured text would apply (Clarke et al., 1995; Sacks-Davis et al., 1997). However, it is possible to have independent annotations of the same signal (speech or text) which chunk the data differently. As a simple example, the division of a text into sentences is usually incommensurable with its division into lines. Such structures cannot be represented using nested, balanced tags.

The fundamental problem faced by any general purpose query language for linguistic annotations is the navigation of these multiple intersecting hierarchies. In this section we consider two query languages which address this issue.

2.1. The Emu query language

The Emu speech database system [`www.shlrc.mq.edu.au/emu`] (Cassidy and Harrington, 1996; Cassidy and Harrington, 1999) provides tools for creation and analysis of data from annotated speech databases. Emu annotations are arranged into levels (e.g. phoneme, syllable, word), and levels are organized into hierarchies. Emu supports multiple independent hierarchies, such that any specific level may participate in more than one orthogonal structure. An example is shown in Figure 1 (Cassidy and Bird, 2000).

A database of such annotations can be searched using the Emu query language. The language has primitives for sequence, hierarchy and "association", as illustrated below.

`[Phonetic=a|e|i|o|u]` – matches a disjunction of items on the phonetic level

`[Phonetic=vowel -> Phonetic=stop]` – matches a sequence of `vowel` followed immediately by `stop`.

`[Word!=dark ^ Phoneme=vowel]` matches an word not labelled `dark` immediately dominating `vowel`.

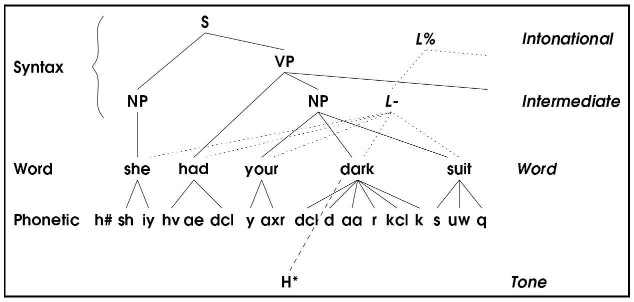

Figure 1: Intersecting Hierarchies in Emu

`[Word!=x => Tone=H*]` Find any word associated[1] with a `H*` tone

Note that the language lacks a wildcard, and `Word!=x` serves this purpose in the absence of any actual word `x`.

More complex queries are built up using nesting. There is no (non-atomic) disjunction or negation in the language. An example of a nested query follows; here, the query finds any syllable dominating a stop that precedes a vowel which is associated to a high tone.

```
[Syllable=S ^
    [Phonetic=stop ->
        [Phonetic=vowel => Tone=H*]]]
```

Cassidy has shown how expressions of this query language can be translated into a first-order query language, in this case, SQL (Cassidy, 1999).

In the Emu query language, the dominance relation is symmetric. (A separate type hierarchy is used to order the levels.) This property makes it possible to navigate a path through multiple hierarchies without using variables. For example, The following expression finds an `NP` which dominates a word `dark` that is dominated by an intermediate phrase that bears an `L-` tone.[2]

```
[ syntax=NP ^ [ word=dark ^ intermediate=L- ]]
```

These expressions correspond to the "where" clause of a conventional query language. The Emu query language lacks an explicit "select" clause. Rather the selected material is the left-most element of the where clause, by default, or else the single element distinguished with a hash prefix. The query result is a column of these elements, and this is typically processed with an external statistics package.

2.2. The MATE query language

The MATE project is developing standards and tools for annotating spoken dialogue corpora [`mate.nis.sdu.dk`]. Like Emu, MATE supports intersecting hierarchies; Figure 2 illustrates four hierarchies built over the same dialogue transcript (Carletta and Isard, 1999). These hierarchies happen to intersect at their fringe, however this need not be the case.

[1]This "association" can have either a temporal interpretation as overlap (Bird and Klein, 1990) and an atemporal interpretation as some essentially arbitrary binary relation; both interpretations are encompassed by our model.

[2]We are grateful to Steve Cassidy for providing this example.

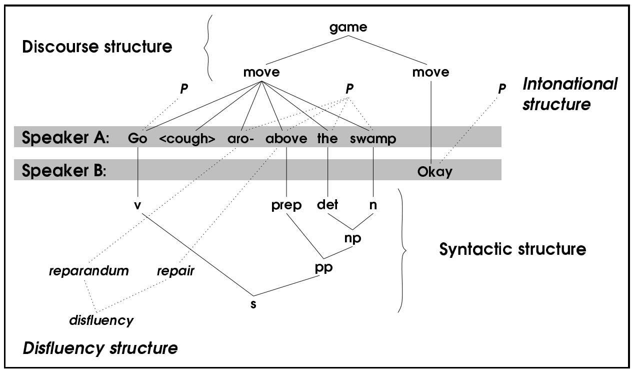

Figure 2: Intersecting Hierarchies in MATE

MATE uses XML to represent these structures. Each node in Figure 2 corresponds to an XML element, and the node labels correspond to an attribute of the element or its content. For example, `swamp` could be represented as `<word id="A6" num="sing">swamp</word>`, and `np` could be represented as `<phrase type="np"/>`. In the query language (Mengel et al., 1999), we can pick out these elements with the following expressions:

```
($w word); $w.orth ~ "swamp"
($p phrase); $p.type ~ "np"
```

Hierarchical relationships, like the one between `game` and `move` or between `move` and `swamp`, are represented using nesting of XML elements or by hyperlinks. The query language has a transitive dominance relation `^` which navigates down through nested structures and hyperlinks. For example, we can find noun phrases dominating the word "swamp" with the expression:

```
($p phrase) ($w word);
  ($p.type ~ "np")
  && ($w.orth ~ "swamp")
  && ($p ^ $w)
```

Each element spans an extent of textual material, and the query language supports a variety of temporal comparisons on these extents, reminiscent of Allen's temporal relations (Allen, 1983). So long as two hierarchies intersect at their terminals (and not at non-terminals) then their non-terminals will be comparable using these temporal expressions. However, the language directly supports queries on intersecting hierarchies. For example, we can find a word which is simultaneously a repair and a preposition, where `1^` is the immediate dominance relation:[3]

```
($w word) ($ph phrase) ($r repair) ($d disfluency);
  ($r 1^ $w) && ($ph 1^ $w)
   && ($ph type ~ "prep") && ($d 1^ $r)
```

Unlike the Emu query language, the formal and computational properties of the MATE query language, vis-à-vis relational and semistructured query languages, are unexplored.

This concludes our brief survey of query languages for annotated speech. Other query languages exist; these two were chosen because of their interesting approach to the problem of intersecting hierarchies.

[3]We thank David McKelvie for furnishing this example.

3. Annotation Graphs

Annotation Graphs were presented by Bird and Liberman as follows. Here we consider just the so-called "anchored" variety.

Definition 1 *An* ***anchored annotation graph*** G *over a label set* L *and timelines* $\langle T_i, \leq_i \rangle$ *is a 3-tuple* $\langle N, A, \tau \rangle$ *consisting of a node set* N*, a collection of arcs* A *labeled with elements of* L*, and a time function* $\tau : N \rightharpoonup \bigcup T_i$*, which satisfies the following conditions:*

1. $\langle N, A \rangle$ *is a labeled acyclic digraph containing no nodes of degree zero;*
2. *for any path from node* n_1 *to* n_2 *in* A*, if* $\tau(n_1)$ *and* $\tau(n_2)$ *are defined, then there is a timeline* i *such that* $\tau(n_1) \leq_i \tau(n_2)$*;*
3. *If any node* n *does not have both incoming and outgoing arcs, then* $\tau : n \mapsto t$ *for some time* t*.*

Note that annotation graphs may be disconnected or empty, and that they must not have orphan nodes. It follows from the above definition that every node has two bounding times, and we will make use of this property later. It also follows from the definition that timelines partition the node set.

The formalism can be illustrated with an application to a simple speech database, the TIMIT corpus of read speech (Garofolo et al., 1986). This database contains recordings of 630 speakers of 8 major dialects of American English, each reading 10 phonetically rich sentences [`www.ldc.upenn.edu/Catalog/LDC93S1.html`]. Figure 3 shows part of the annotation of one of the sentences. The file on the left contains word transcription, and the file on the right contains phonetic transcription. Part of the corresponding annotation graph is shown underneath. Each node displays the node identifier and the time offset (in 16kHz sample numbers). The arcs are decorated with type and label information. The type `W` is for words and the type `P` is for phonetic transcriptions.

Observe that all the nodes in Figure 3 have time values. This need not be the case. For example, in the CALLHOME telephone speech corpus [`www.ldc.upenn.edu/Catalog/LDC96S46.html`], times are only available for speaker-turn boundaries (see Figure 4).

```
train/dr1/fjsp0/sa1.wrd:   train/dr1/fjsp0/sa1.phn:
2360 5200 she              0 2360 h#
5200 9680 had              2360 3720 sh
9680 11077 your            3720 5200 iy
11077 16626 dark           5200 6160 hv
16626 22179 suit           6160 8720 ae
22179 24400 in             8720 9680 dcl
24400 30161 greasy         9680 10173 y
30161 36150 wash           10173 11077 axr
36720 41839 water          11077 12019 dcl
41839 44680 all            12019 12257 d
44680 49066 year           ...
```

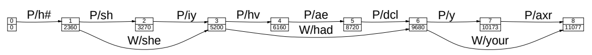

Figure 3: TIMIT Annotation Data and Graph Structure

Annotations expressed in the annotation graph data model can be trivially recast as a set of relational tables (Cassidy and Bird, 2000), just as can be done for semistructured data (Florescu and Kossmann, 1999). We employ three relations: *arc*, *time* and *label*. The arc relation is a four-tuple containing an arc id, a source node id, a target node id, and a type. The time relation maps (some of) the node ids to times. The label relation maps the arc ids to labels.

Figure 5 gives an instance of this schema for the TIMIT data of Figure 3 (enriched with the information shown in Figure 1. The names of key attributes are underlined. Figure 6 shows the graph representation for this data. Note that intersecting hierarchies find a natural expression in this model.

4. Some Example Queries

Interesting cases for query are those that involve more than one of these primitives. Here are some simple queries to select subsets of the data.

1. Find word arcs whose phonetic transcription contains a 'd' and ends with a 'k'.
2. Find phonetic arcs which immediately precede a vowel that overlaps a high tone.
3. Find words dominating a vowel which overlaps a high tone.

These queries can be interpreted against the fragment shown in Figure 7.

Such queries have a first-order interpretation in graphlog (Consens and Mendelzon, 1990). We employ a datalog syntax and the relations in Figure 5. We begin by defining some auxiliary relations.

First we define a path relation that is sensitive to arc types. Two nodes `X` and `Y` are connected by a path of type `T` if there is a sequence of zero or more arcs, all of type `T`, beginning at `X` and ending at `Y`.

```
path(X,X,T) :- arc(_,X,_,T)
path(X,X,T) :- arc(_,_,X,T)
path(X,Y,T) :- arc(_,X,Z,T), path(Z,Y,T)
```

An arc `A` "structurally includes" an arc `B` if there is a path from the start node of `A` to the start node of `B`, and a path from the end node of `B` to the end node of `A`.

```
s_incl(A, B) :- arc(A, X1, Y1, _),
                arc(B, X2, Y2, _),
                path(X1, X2, _),
                path(Y2, Y1, _)
```

Finally, an arc `A` "temporally overlaps" an arc `B` if the start node of `A` precedes the end node of `B`, and the start node of `B` precedes the end node of `A`. (See section 6. for details of the precedence relation.)

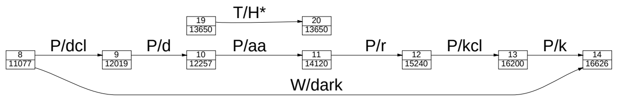

Figure 7: An Annotation Graph Fragment

```
962.68 970.21 A: He was changing projects every couple of weeks and he
  said he couldn’t keep on top of it. He couldn’t learn the whole new area
968.71 969.00 B: %mm.
970.35 971.94 A: that fast each time.
971.23 971.42 B: %mm.
972.46 979.47 A: %um, and he says he went in and had some tests, and he
  was diagnosed as having attention deficit disorder. Which
980.18 989.56 A: you know, given how he’s how far he’s gotten, you know,
  he got his degree at &Tufts and all, I found that surprising that for
  the first time as an adult they’re diagnosing this. %um
989.42 991.86 B: %mm. I wonder about it. But anyway.
991.75 994.65 A: yeah, but that’s what he said. And %um
994.19 994.46 B: yeah.
995.21 996.59 A: He %um
996.51 997.61 B: Whatever’s helpful.
997.40 1002.55 A: Right. So he found this new job as a financial
  consultant and seems to be happy with that.
1003.14 1003.45 B: Good.
```

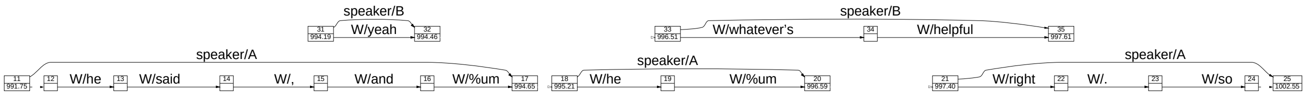

Figure 4: CALLHOME Telephone Speech Data and Graph Structure

Arc

<u>*A*</u>	*X*	*Y*	*T*
1	0	1	P
2	1	2	P
3	2	3	P
4	3	4	P
5	4	5	P
6	5	6	P
7	6	7	P
8	7	8	P
9	8	9	P
10	9	10	P
11	10	11	P
12	11	12	P
13	12	13	P
14	13	14	P
15	14	15	P
16	15	16	P
17	16	17	P
19	3	6	W
20	6	8	W
21	8	14	W
22	14	17	W
23	1	18	S
24	3	18	S
25	1	3	S
26	3	6	S
27	6	17	S
28	1	17	Imt
29	1	18	Itl
30	1	19	T
31	19	20	T

Time

<u>*N*</u>	*T*
0	0
1	2360
2	3270
3	5200
4	6160
5	8720
6	9680
7	10173
8	11077
9	12019
10	12257
11	14120
12	15240
13	16200
14	16626
15	18480
16	20685
17	22179

Label

<u>*A*</u>	*L*
1	h#
2	sh
3	iy
4	hv
5	ae
6	dcl
7	y
8	axr
9	dcl
10	d
11	aa
12	r
13	kcl
14	k
15	s
16	uw
17	q
18	she
19	had
20	your
21	dark
22	suit
23	S
24	VP
25	NP
26	V
27	NP
28	L-
29	L%
30	0
31	H*

Figure 5: The Arc, Time and Label Relations

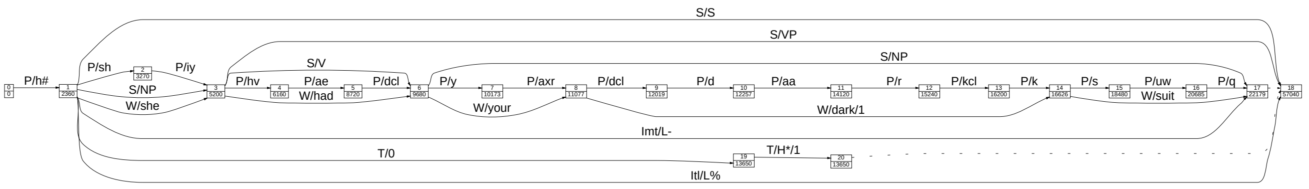

Figure 6: Annotation Graph for Extended TIMIT Example

```
ovlp(A, B) :- arc(A, X1, Y1, _), arc(B, X2, Y2, _),
              time(X1, X1t), time(X2, X2t),
              time(Y1, Y1t), time(Y2, Y2t),
              X1t ≤ Y2t, Y1t ≤ X2t
```

Now we can provide translations for the three queries listed above.

1. Find word arcs whose phonetic transcription contains a 'd' and ends with a 'k'. We assume a relation path/3 which is the transitive closure of arc/4.

```
ans(A) :- arc(A, X, Y, word),
          path(X, X1, phonetic),
          arc(A1, X1, X2, phonetic), label(A1, d),
          path(X2, X3, p),
          arc(A2, X3, Y, phonetic), label(A2, k)
```

2. Find phonetic arcs which immediately precede a vowel that overlaps a high tone:

```
ans(A) :- arc(A, X, Y, phonetic),
          arc(A1, Y, Y1, phonetic), label(A1, [aeiou]),
          arc(A2, Z, Z1, tone), label(A2, h*)
          ovlp(A1, A2)
```

3. Find words dominating a vowel which overlaps a high tone:

```
ans(A) :- arc(A, _, _, word),
          arc(A1, _, _, phonetic), label(A1, [aeiou]),
          arc(A2, _, _, tone), label(A2, h*),
          s_incl(A, A1), ovlp(A1, A2)
```

While it is possible to give queries a first-order interpretation, the language is quite cumbersome, and we seek a more natural way to describe annotation graphs.

5. Query Syntax

In this section we introduce a query syntax which provides first an abbreviated notation for the queries expressed previously in datalog. Most importantly, the syntax allows us to recognize certain crucial optimizations.

5.1. Queries over arc data

The fundamental unit on which our query language is built is the arc. We form the join of the arc and label relations from Figure 5 and adopt names for our attributes. A query that finds the arc identifiers, types and labels of all edges in timeline `tl1` is shown below:

```
select ans(E,T,L)
where [id: E, type: T, label: L] <- tl1
```

We follow the datalog convention of using uppercase symbols for variables and lowercase symbols for constants. The notation `[id: E, type: T, label: L]` is used for arcs and describes a *arc pattern*: it is matched against the arcs in the timeline `tl1` and binds the variables `E,T,L` for each match to the arc data in the timeline. For each such match it constructs a tuple `ans(E,T,L)` in the output. Arc patterns may contain constants, e.g. `[id: E, type: word, label: L]` and there is no constraint on their width. In this sense they are "ragged" or "semistructured" tuples.

```
[id: E, start: X, end: Y, type: T, name: N,
  xref: X, lex-id: L, annotator: SB]
```

Since attributes are distinguished by name rather than position, it is safe (and often convenient) to omit them when we do not need to constrain their value, or bind a variable.

To query over a collection of timelines `timit` we use cascaded bindings:

```
select ans(E,L)
where  TL <- timit
       [id: E, start: X, end: Y, label: L, type: word] <-TL
       time(Y) - time(X) < 8000
```

This selects the edge identifiers and labels (the names of the words) from all words in the timit corpus of a suitably short duration.

The form of this query follows a standard syntax for semistructured query languages (see (Abiteboul et al., 2000)). We shall concentrate here on the development of a syntax for patterns that specify paths and assume a standard syntax, e.g. `select ans(E,L)`, for returning results of the query.

5.2. Path patterns

Each arc has a start and end node. We can specify two adjacent arcs by requiring the start node of one arc to be the end node of another

```
[id: E1, start: X, end: Y, type: T1, label: L1] <- db
[id: E2, start: Y, end: Z, type: T2, label: L2] <- db
```

In this fashion we can specify any sequence of arcs. However we shall use an abbreviated syntax `[ ... ].[ ... ]` to specify the concatenation of edges, that is, the dot is an associative pattern concatenation operation. Thus the previous pair of patterns binds the same variables as the following single pattern

```
[id: E1, start: X, end: Y, type: T1, label: L1] .
[id: E2, end: Z, type: T2, label: L2] <-  db
```

Within edge patterns we also allow arbitrary predicates. For example: `[type: T, T = word or T = ph]`, `[start: X, stop: Y, time(Y) - time(X) > 200]`. Predicates may also use attribute names as values. For example, `[type: word]`, `[type: X, X = word]`, `[type=word]` are equivalent.

A sequence of arcs (phonemes, syllables, phrases, etc) is represented in our model using a concatenated sequence of arc patterns. To specify path patterns of arbitrary length we also allow arbitrary regular expressions on arcs. An arbitrary path of word arcs is represented by `[type = word]*` and an arbitrary path of word or phoneme arcs by `([type = word]|[type = phoneme])*`. Care must be taken in interpreting variables inside a Kleene * or a union. The rule is that such variables must be bound elsewhere in the program. We cannot bind variables inside a union or Kleene *. Thus `[type: T]*` is illegal.

Suppose we have a path pattern `[type: word]*` and we want to refer to the first node on the path. The pattern `[start: X, type: word]*` is illegal. (Even if it were legal this pattern could only only match paths of length 0 or 1.) To allow the binding of nodes outside of an edge pattern we take single variables in the sequence to

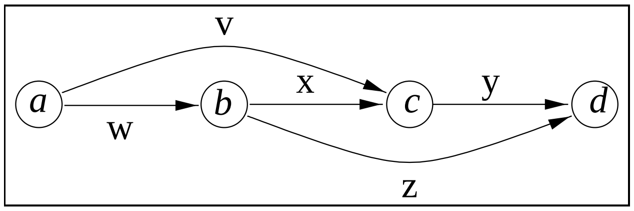

Figure 8: An Annotation Graph whose Description Requires Variables

denote nodes. For example, `X.[type: T].Y` is equivalent to `[start: X, type: T, end: Y]`. Moreover, `X.[type: word]*.Y` binds `X` to the first node and `Y` to the last node on a path of `word` arcs. Now consider the following example:

```
X.[type = parse, label = sentence].Y <- db           (a)
X.[type = word]*.[type = word, label = opera]
 .[type = word]*.Y <- db
```

This matches the start and end node of any sequence that contains the word `opera`. Another possibility is shown below.

```
X.[type = parse, label = sentence].Y <- db           (b)
X'.[type = word, label = opera].Y' <- db
time(X) <= time(X') and time(Y') <= time(Y)
```

However, (a) and (b) are *not* equivalent queries.

One might think, from example (a) above, that one could dispense with node variables by having a parallel composition operator. It turns out that there are many situations where this is impossible. The simplest instance is shown in Figure 8.

The annotation graph in Figure 8 cannot be uniquely described using parallel and serial composition. Instead, we need a set of expressions as follows:

```
A.[label: W].B.[label: X].C.[label: Y].D <- db
A.[label: V].C <- db
B.[label: Z].D <- db
```

5.3. Arbitrary predicates on arcs

The bracket notation for describing arcs can also enclose arbitrary predicates. Predicates expressing the (temporal) overlap or inclusion of edges are particularly useful. Example (b) above may be expressed as.

```
[id: E, type = parse, label = sentence] <- db
[type = word, label = opera, subinterval(E)] <- db
```

Note that `subinterval(E)` can be thought of as a “method” of the edge, that is called when the pattern is matched.

5.4. Abbreviations

The preceding syntax is quite general; it has little to do with the specific conventions of linguistic data. Paths typically, though not always, follow the same type. Labels are also special. We propose the following syntactic sugar. (The proposal is tentative, all sorts of variations are possible).

Given a database of arcs *db*, the notation `db/t` restricts the database to those arcs of type t. Also the notation `:L` is an abbreviation for `label: l`. For example, `X.[:L].Y <- db/word` is shorthand for `X.[label: L, type: word].Y <- db`. Using this, example (a) becomes:

```
X.[:sentence].Y <- db/parse                          (a′)
X.[]*.[:opera].[]*.Y <- db/word
```

5.5. Horizontal path expressions

Find words with `c.*t.*` (our first query)

```
X.[].Y <- db/word
X.[:c].[]*.[:t].[]*.Y <- db/ph
```

Here's a harder case, with a variable inside the scope of a Kleene star. The predicate `ovlp(E)` is an “overlap” predicate.

```
X.[].Y <- db/word
[id: E] <- db/background
X.[:c].[ovlp(E)]*.[:t].[]*.Y <- DB/ph
```

In this section we have paid little attention to the output of a query. From the introductory examples, it should be clear that it is straightforward to construct a set of tuples in the same sense that datalog constructs a set of tuples. It is also possible to extend the syntax to express the construction and augmentation of annotation graphs. The details will be described elsewhere

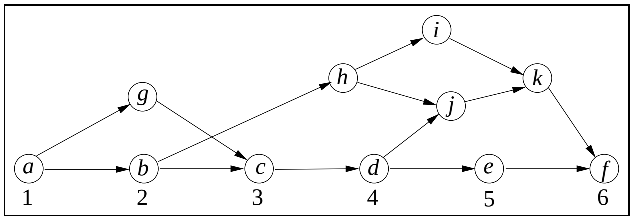

Figure 9: A precedence graph

6. Optimization: exploiting quasi-linearity

In the previous sections we developed a query language for annotation graph data and showed how an analysis of that language might help - in many practical cases – to lead to tractable implementations. Here we show how we can exploit the “almost sequential” notion of annotation graph data to support these implementations. In particular, we will show how to use the underlying temporal order to select a small fragment of the input data that will fit into main memory, bypassing many of the database optimization issues.

Consider the example in Figure 9. It shows a collection of nodes, where nodes a-f are timed and the rest are untimed. All the untimed nodes are linked by arcs to other nodes. In order to extract those portions of the database that are needed to answer a query, we will typically need to find efficiently all arcs contained in some arc or all arcs that might overlap some arc. Such queries can be answered by computing the transitive closure TA of the arc relation, but this is likely to be an expensive proposition ($O(n^2)$ in the number of nodes). An alternative is to store the two relations below.[4] The relation *time* contains, for every node n, the maximum time *ante* of a timed node that precedes n

[4]Our approach has similarities with Allen's “reference intervals” (Allen, 1983).

time

node	timeline	ante	post
a	T_1	1	1
b	T_1	2	2
c	T_1	3	3
d	T_1	4	4
e	T_1	5	5
f	T_1	6	6
g	T_1	1	3
h	T_1	2	6
i	T_1	2	6
j	T_1	4	6
k	T_1	4	6

TA'

source	target
h	i
h	j
j	k
i	k
h	k

Figure 10: The Time and TA Relations

and the minimum time *post* of a timed node that precedes n. If n is itself timed, the *ante* and *post* agree.[5] It is a consequence of the definitions in section 3. that these times always exist. (Every node is bounded by some pair of timed nodes.) Note that *node* is a key for the *time* relation, and we shall refer to the attributes *ante* and *post* functionally, as *ante*(n) and *post*(n).

The relation TA' is defined by $TA' = \{(m,n) | TA(m,n) \wedge \textit{post}(m) > \textit{ante}(n)\}$. This means that the precedence relation TC can be reconstructed by the query:

$$TC(m,n) :- \textit{post}(m) < \textit{ante}(n) \vee TA'(m,n)$$

With indexes on *ante*, *post*, and (source, target), this predicate can be efficiently computed.

The point of this decomposition is that we expect the relation TA' to be relatively small. For example, in the Switchboard database (Godfrey et al., 1992), the maximum size of TC for any timeline is approximately 1.9 million, while while the sizes of *time* and TA' are, for this timeline, $1,992$ and $10,585$ respectively.[6] Throughout the whole database, the largest value of TA' was $15,286$. Evidently the decomposed representation will easily fit into main memory, while keeping TC in main memory may pose problems.

Finally, let us put together the ideas of the last two sections. Consider example (a) of the previous section. The important point is that all nodes are bounded by a `sentence` arc. This suggests the following technique:

- Repeatedly match X.`[type = parse, label = sentence]`.Y
- For each match, obtain $X' = \textit{ante}(X)$ and $Y' = \textit{post}(Y)$
- Restrict the arc relation to arcs bounded by (X', Y') (use an index that supports range searches)
- Perform the query on the restricted relation (main memory evaluation should be possible)

[5]Some saving in space could be achieved by having a separate relation for the timed nodes.

[6]This computation is based on the version of Switchboard data that is marked with time information at turn boundaries only. Given an n-word turn, the size of the transitive precedence relation is approximately $n^2/2$.

7. Conclusions

Like semistructured data, annotation graphs have a natural representation in terms of nodes and arcs. A key feature of annotation graphs is that the arcs are organized into a quasi-linear flow in the horizontal direction. As in the case of semistructured data, we seek a natural query language for accessing and transforming this data.

This paper has described progress on a query language for annotation graphs. Path patterns and some abbreviatory devices provide a convenient way to express a wide range of queries. We exploit the quasi-linearity of annotation graphs by partitioning the precedence relation, and we believe that this will enable efficient temporal indexing of the graphs.

In ongoing work we are exploring hybrid structures and languages which would permit both the vertical and horizontal perspectives on semistructured data to co-exist. On this view, a horizontal path expression could be embedded inside a vertical path expression, or vice versa.

8. References

Abiteboul, Serge, Peter Buneman, and Dan Suciu, 2000. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann.

Allen, James F., 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–43.

Bird, Steven and Ewan Klein, 1990. Phonological events. *Journal of Linguistics*, 26:33–56.

Bird, Steven and Mark Liberman, 1999. A formal framework for linguistic annotation. Technical Report MS-CIS-99-01, Department of Computer and Information Science, University of Pennsylvania. [xxx.lanl.gov/abs/cs.CL/9903003], expanded from version presented at ICSLP-98, Sydney, revised version to appear in *Speech Communication*.

Buneman, Peter, Susan B. Davidson, Gerd G. Hillebrand, and Dan Suciu, 1996. A query language and optimization techniques for unstructured data. In H. V. Jagadish and Inderpal Singh Mumick (eds.), *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. ACM Press.

Carletta, Jean and Amy Isard, 1999. The MATE annotation workbench: user requirements. In *Towards Standards and Tools for Discourse Tagging – Proceedings of the Workshop*. Somerset, NJ: Association for Computational Linguistics.

Cassidy, Steve, 1999. Compiling multi-tiered speech databases into the relational model: experiments with the Emu system. In *Proceedings of the 6th European Conference on Speech Communication and Technology*. [www.shlrc.mq.edu.au/emu/eurospeech99.shtml].

Cassidy, Steve and Steven Bird, 2000. Querying databases of annotated speech. In *Proceedings of the Eleventh Australasian Database Conference*. Los Alamitos, CA: IEEE Computer Society.

Cassidy, Steve and Jonathan Harrington, 1996. Emu: An enhanced hierarchical speech data management system. In *Proceedings of the Sixth Australian International*

Conference on Speech Science and Technology. [www.shlrc.mq.edu.au/emu/].

Cassidy, Steve and Jonathan Harrington, 1999. Multi-level annotation of speech: an overview of the emu speech database management system. Manuscript.

Clarke, Charles L. A., G. V. Cormack, and F. J. Burkowski, 1995. An algebra for structured text search and a framework for its implementation. *Computer Journal*, 38:43–56.

Consens, Mariano P. and Alberto O. Mendelzon, 1990. The G+/graphlog visual query system. In *SIGMOD Conference*.

Deutsch, Alin, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu, 1998. XML-QL: A query language for XML. [www.w3.org/TR/NOTE-xml-ql/].

Florescu, D. and D. Kossmann, 1999. A performance evaluation of alternative mapping schemes for storing XML data in a relationl database. Manuscript.

Garofolo, John S., Lori F. Lamel, William M. Fisher, Jonathon G. Fiscus, David S. Pallett, and Nancy L. Dahlgren, 1986. *The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus CDROM*. NIST. [www.ldc.upenn.edu/lol/docs/TIMIT.html].

Godfrey, J. J., E. C. Holliman, and J. McDaniel, 1992. Switchboard: A telephone speech corpus for research and develpment. In *Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing*, volume I. [www.ldc.upenn.edu/Catalog/LDC93S7.html].

Mengel, Andreas, Ulrich Heid, Arne Fitschen, and Stefan Evert, 1999. Specification of coding workbench: Improved query language (q4m). Technical Report MATE Deliverable D3.1, Stuttgart: Institut für Maschinelle Sprachverarbeitung. [www.ims.uni-stuttgart.de/projekte/mate/q4m/].

Quass, D., A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom, 1995. Querying semistructure heterogeneous information. In *International Conference on Deductive and Object Oriented Databases*.

Sacks-Davis, Ron, Tuong Dao, James A. Thom, and Justin Zobel, 1997. Indexing documents for queries on structure, content and attributes. In *International Symposium on Digital Media Information Base*.