Text Databases: A Survey of Text Models and Systems

Arjan Loeffen

Computer & Humanities, Faculty of Arts, University of Utrecht, The Netherlands
Achter de Dom 22-24, 3512JP Utrecht (LOEFFEN@LET.RUU.NL)

Abstract. Text models focus on the manipulation of textual data. They describe texts by their structure, operations on the texts, and constraints on both structure and operations. In this article common characteristics of machine readable texts in general are outlined. Subsequently, ten text models are introduced. They are described in terms of the datatypes that they support, and the operations defined by these datatypes. Finally, the models are compared.

1. Introduction

Texts are converted to electronic representations for all kinds of purposes. The most central operation performed on these texts is that of retrieval. Most operations are founded on elaborate text querying, the results of which may be used for comparisons, sorting, numerical computing, printing, editing, annotating, etcetera, as well as for additional retrieval operations. In text-based information systems the electronic texts are required to be available in a form that allows for dedicated access. Access of texts on a word-for-word basis ignores important parts of the information present in the source text. Query facilities should be powerful, and allow for a more complete access to the electronic source(s). This includes structural, linguistic, associative and linear access. If the retrieval part of the system is powerful enough, any kind of text accessing system may be defined 'on top of' this system.

The individual characteristics of the texts that are ultimately stored for retrieval cannot be predicted. However, these texts may be expected to conform to a formal *model* of text. The way 'text' is thus abstracted into a model may be reflected in the accessing language, that applies to the formal aspects of these texts. The power of the accessing language is proportional to the extent the information is conveyed in the electronic text.

Such a text model consists of three elements: the way text features are abstracted, the operations that are required for accessing the information in store, and the constraints in text representation and access. In this article these three aspects are introduced. Next, several existing text models are described shortly, and finally all models are discussed in the light of the three aspects of a text model mentioned.

2. Background

Text and factual databases. There's a clear difference between the way text directed systems on the one hand, and text extended database systems on the other treat textual sources. In database systems, text is treated as a special kind of field value that calls for some operators that allow the query to locate terms in that text field. However, formalizations of textual relations in these factual database systems have received little attention from the database community. Text is mostly a secondary source of information, and document structure, access paths and constraints have not been defined on an abstract level:

> *"Most commonly, retrieval systems deal only with two kinds of textual objects: the word, and the document containing it — any intermediate structure is left unrepresented, and therefore inaccessible. [..] The relatively few systems that represent any structure beyond the document and the word do so by either hard-copying a particular structure [..] or by adapting a traditional structured database approach, treating textual objects as though they were fields."* [DEERW92:128]

More structure-oriented text directed models may result in relevant tools in themselves, but could also be adapted by both text-based (TB) and text extended (DB+text) system models. They will not only set a context for source oriented research, but also take part in more industrial applications. Such models are treated in this section.

Characteristics of electronic text. All text models abstract the nature of text in general. Programs based on the model will therefore operate on machine readable sources that may be expected to share most characteristics of their physical counterparts. Text models should address these characteristics in a consistent way, as these are features shared by all machine readable texts. Some of these are mentioned below.

- Texts are written in *natural language*. Therefore, character sets may differ between texts, the concept of 'word' may differ between languages and users, sentence boundaries are sometimes hard to determine, etc.
- As the models deal with electronic text, the *encoding* strategies used to convey characteristics of the original text are of high importance. More and more, texts are exchanged in an encoded form. The most prominent language in this respect is the Standard Generalized Markup Language (SGML, [ISO8879]), though numerous other encoding schemes are in use. It is fictitious to assume the texts will be rendered in a bare form, i.e. without structural and informational codes.
- Texts are complex objects. *Complexity* may for example take the form of hierarchical and linear relations between components, and of references from one component to another. An important structural feature of texts is that in-line components, such as a footnote or citation, may occur anywhere in the text. Next, text hierarchies are based on containment relations. This means that any operation on the 'super' text also applies to the 'sub' text. Containment should be an intrinsic part of the model of text structure.
- Strongly related to the previous, the text system should also be able to determine the *basic components*, i.e. what parts of the text may be treated as a unit for editing, moving, linking and so on. This decision lies not only on the level of textual representations (inversion versus linear text scanning), but also on the level of text structure.
- Texts and subtexts are usually augmented with meta-information of diverse nature (*attributes*). This may take the form of labels attached to text sections, complete documents, or simple tags 'pinned' on a single word.

- One and the same text may contain *parallel* text structures. For example, a document may be structured by physical characteristics (pagination, rendition), parallel with its logical structure. Any operation that concerns one structure influences the structure of the other levels.
- Texts may be *mutable* by nature or use. In that case updates may violate the integrity of the information in store.

Text models. If we are to select some text models, we should first determine what we understand as a model. On this issue, [TAGUE91] states:

> *"The purpose of a formal system model is to describe the common features of a set of systems which have been developed for similar problems. The model will explain the structure and processes of these systems, and clarify their general, as opposed to specific, characteristics. The components of a model must include the kinds of entities, relationships and transformations or operations which form a part of the system which it is intended to describe. A complete model will contain a representation of all components in any system of the kind referenced by the model."*

It is assumed that textual data are modelled if three aspects are covered: the text structure, operations that are allowed and relevant for text handling, and constraints that apply on both structure and operations. These may each be separated in a *physical* description and a *logical* description. We should however focus on the latter if not describing a particular implementation. The aspects should align with text characteristics as mentioned earlier.

- *Structure*. The model will show how textual data are interrelated such that a correct representation of the source may be given. This does not mean that particular logical text descriptions are presented. The model should allow the user to specify the text type (all structural features, including different information levels) in a uniform fashion.
  Structural features are abstracted into a definition scheme that describes all possible text structures. This usually takes the form of a *text grammar*. Subsequently, text are classified (a type is assigned to it), and the text is sectioned into manageable parts. Thus *schema's* in factual database systems would be counterparted by *grammars* in a textual context.
- *Operations*. The model focuses on how operations may address those objects that are described by the grammar. Again, particular interpretations of such objects are not described. The model will only describe the basic relations between textual objects (hierarchical, linear, referencing) and the way such relations may be defined and queried.
- *Constraints*. Logical constraints are imposed on text manipulations. This includes restrictions in cross-referencing, removal of structural objects, depth of nesting, and so on. Purely on the operational side, constraints may be formulated such as the way a footnote should be accessed, if annotations may be exported without the annotated text section to go with it, and so on. Any model will show how such constraints may be formulated — it will however *not* define these particular constraints.

In the following some text models are presented shortly. Most of the models are actually used in implementing a text handling application. Structural characteristics of text are most extensively defined by the SGML standard ([ISO8879]) and the ODA standard ([ISO8613]). The models address both document structure and behavior, but do not (completely) align with one or both of these standards.

The models are described (where possible) by the datatypes that are vital to the model, and the operations and constraints that are defined upon these datatypes. The description of the models in terms of datatypes allows for a clear distinction between data, procedures and constraints. I like to stress that such types are *deferred* from the references on the models, and are therefore not always available as a supported type.

3. TDM

Introduction. As the relational model is widely in use, and textual sources are to be incorporated, it seems reasonable to try to extend the relational operator set on these textual objects. This is the approach taken by the TDM model. [DESAI86] describes a *text data model* (TDM) based on nonfirst normal form (NFNF). The aim of the TDM is to extend the operations defined on textual fields in a NFNF relational model. TDM assumes the relational model to pass the text fields as TEXT instances; it returns a success code (succeeds or fails) as the evaluation result. For instance, the following NFNF relational expression will succeed for all tuples in the database that have the text `"May"` in the `ABSTRACT` field:

```
retrieve  TITLE
where     ABSTRACT\"May" „ ABSTRACT
```

Such queries are relatively simple, but assume that the `where` clause activates a subprocess that retrieves information of the non-normalized `ABSTRACT` field. This is the focus of the TDM model.

Objects. TDM introduces four basic datatypes for text processing, and the operations that apply. TEXT is a recursive relation of CHARACTER (ordered). A *text value* (constant) is enclosed between matching characters, eg. `"text"`, `Mother textM`. An (UN)ORDERED SET is an (un)ordered list of unique characters or texts. The ORDERED SET constants are denoted by `(a,b,c)`. The UNORDERED SET is denoted by `{a,b,c}`. Sets do not collect sets nor expressions. Furthermore, a **struct** is supported to model dependent substructures.

Operations. Special operators are applied to the instances. The most basic operator that applies to TEXT is the search operator •, e.g.:

```
r ‹ "John" • "h"
```

assigns `"h"` to `r` if `"h"` is contained in `r`. *Text expressions* (that operate on text) produce TEXT or (UN)ORDEREDSET instances.

Concatenation (+) is defined on TEXT and on UNORDEREDSET. For *text concatenation*, the argument may be any text expression, and therefore may include ordered sets (unordered sets are in this case not supported):

```
"Ope" + "Admiration" • "ration"
    returns "Operation"
"Ope" + "Admiration" • ("ration", "mi")
    returns "Operationmi"
```

This last expression is transduced to `"Ope" + "Admiration" • "ration" + "admiration" • "mi"` which is evaluated, as all expressions in TDM, from left to right.

Unordered set concatenation is the regular set union, defined on UNORDEREDSET only.

Removal (-) is defined over TEXT and UNORDEREDSET. *Text removal* is complementary to concatenation. It removes the first text occurrences from a text instance.

```
"Operation specification" - "ion"
    returns "Operat specification"
```

Unordered set removal is the regular set difference operation. The same question as formulated above applies here.

Copy-extraction (¯) is defined over TEXT, and produces an ordered set with two elements: the first result is due to concatenation, the second to subtraction. `"Specification "` ¯ `"pe"` returns `("Specification pe", "Scification ")`, which comes down to moving `"pe"` from the start of the subject to the end.

Division (/) is also defined on TEXT. The operation is similar to removal, but is more elaborate. It has two results: the quotient and the remainder. For instance, `"abc" / "b"` returns `("a","c")`, i.e. the subtexts that precede and follow the nominator `"b"` in the denominator `"abc"`, respectively. Two operators `t1/t2` and `t1\t2` are available to select the preceding or following part of the division. The nominator may be an ordered set, for example:

```
"ABCDEFG" / ("B", "D", "F")
    returns ("A","G")
```

The operation is in fact a fragmentation operation, as the elements of the parameter set are used to select each left element in the subject text preceding any element of the unordered set. For example,

```
"You made an error." / {" ", "."}
    returns {"You", "made", "an", "error"}
```

These constructions are typically used to define *text views* that are much like functions in the sense that text expressions may be assigned to them, specifying a *placeholder* variable in both the expression and the text view:

```
punctuation_list < {" ", ",", ".", "?", "!"}
define word_list(t) < t / punctuation_list
```

The text views take on the type of the expression (`word_list` would be of type TEXT), and may be used where this type is accepted.

Other 'datatypes' may be constructed at will using a traditional **struct**. A typical use for this primitive is the collection of text views into a grammar specifying the text structure.

4. P-strings

Introduction. The focus of the *p-string model* of [GONNE87] is *text-dominated databases*, described as 'collections of structured data, predominantly composed of alphabetic characters'. The authors focus on dictionaries, news clippings, legal documents and so on: highly structured data that allow a high degree of normalization, that do not fit easily into tables — the main argument is that these types of 'records' cannot be treated as tuples, but should be handled as trees of information units, described by grammars, and manipulated by tree manipulation functions. The paper presents a model for text-dominated databases as a textual counterpart for relational models for factual data. A variant of this model is presented in [TAGUE91]. [GYSSE89] elaborates on the mathematical fundamentals of grammar based models such as the p-string model. They define the minimal tree manipulations as both an *algebra* (on instances, where the D-tree datatype is central), and as a *calculus* (on collections, which calls for the R-TREE datatype). These operations (8 in total for D-trees) form a complete set for the implementation of the p-string model. However, in this section only the p-string model is described.

Objects. We may recognize several datatypes used in the p-string model. STRING is any sequence of alphabetical characters. The P-STRING is the representation of a parsed string, i.e. a string that has been described by a grammar and is represented accordingly. GRAMMAR, then, is a set of rules that is identified by a non-terminal symbol (which serves as a local root). The grammar is built by a parser generator for context free grammars. Its interface is not described in [GONNE87], in contrast with [GYSSE89], who describe both the structure and the schema behavior of similar grammars. An example of a PAT grammar is given here (these rules are part of an `article` grammar):

```
...
source := 'In: ' compname ' ' vol ' (' nr '), '
        year
compname := '[' shortname shortyear ']'
nr := digit+
vol := digit+
shortname := char+
shortyear := digit digit
year := '19' shortyear
```

Other datatypes are INTEGER and BOOLEAN in the traditional sense, and VECTOR and SET, that collect subtrees as a list or an ordered set (no duplicate subtrees). A third special datatype is PARTITION, a vector with two elements. This is the result of the **partitioned by** operator. Finally, FUNCTIONS are treated as datatypes that implement *processes*. These may succeed and fail, returning booleans that implement operators such as **and** and **or**.

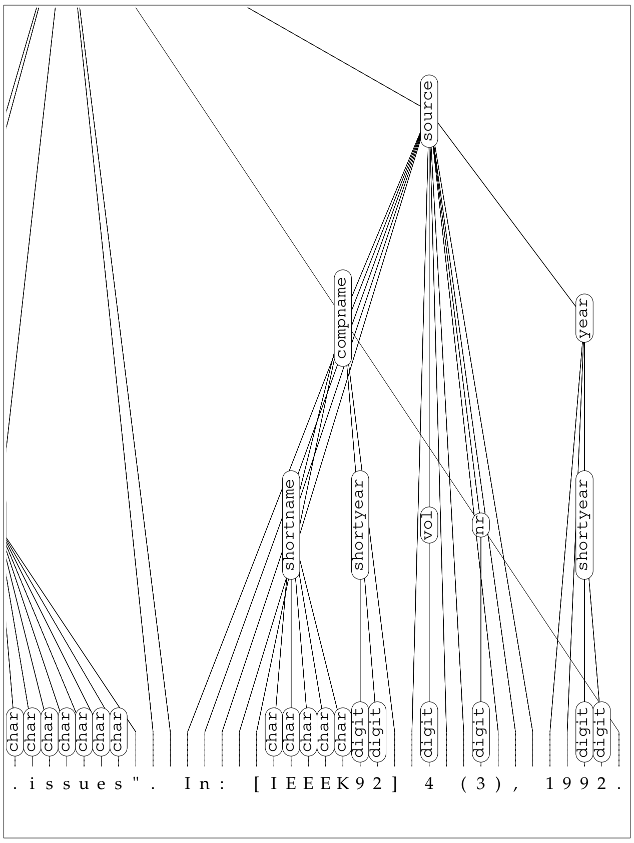

Figure 1 - P-STRING represented as a tree.

Operations. The operation **parsed by** is defined on STRING. It parses a STRING according to a grammar and yields a P-STRING. For example:

```
'...In: [IEEEK92] 4 (3), 1992.'
    parsed by Article
```

returns a P-STRING, the `source` part of which is depicted in Fig. 1.

reparsed by, defined on P-STRING, re-evaluates a P-STRING according to a (SUB)GRAMMAR, identified by a root symbol, i.e. each node labeled by that symbol will be reparsed. For example, suppose the grammar for `source` is modified:

```
source := 'In: ' compname ' ' vol ' (' nr '), '
          year '. (' pg '-' pg ')'
pg := digit+
```

Applying the **reparsed by** operator to `'In: [IEEEK92] 4 (3), 1992. (223-237)'` will yield the p-string displayed in Fig. 2.

These two operations are central to the p-string model. Some other interesting operations are listed below. **in**, **subtrees** and **every** are typical tree-management operations that do not depend on the grammar specification. The **transduced by** operation uses the grammar to create a new p-string. **partitioned by** and **where** are based on the evaluation of FUNCTION instances passed.

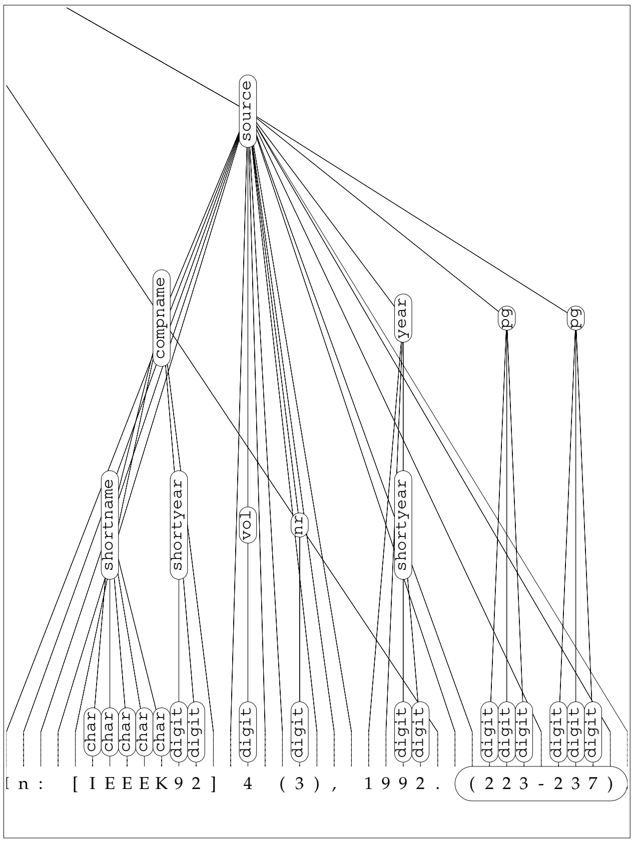

Figure 2 - A reparsed reference.

The **in** operator, defined on P-STRING, takes a symbol and locates the first occurrence of that symbol in the p-string by a depth first, left to right search. The result of the operator is a copy of the subtree. **subtrees**, defined on P-STRING, accepts an argument `P` of type symbol, and returns the immediate subtrees of each node `P` as a VECTOR. **every**, defined on P-STRING, is similar to **in**, but performs a full traversal and collects all subtrees with the node labeled by the symbol given in the VECTOR passed as a parameter. **suppressing**, defined on P-STRING, removes all subtrees that are labeled by the (set of) symbol(s) passed.

transduced by, defined on P-STRING, accepts a p-string and a part of a grammar, and yields the subsection of p-string that conforms to the new grammar. For instance, if we are to define a grammar G' (where G is the original grammar used to parse E) and pass it to E, this will yield a short book description of the kind `result = "Gonnet (1987)"`:

```
G' := {article := surname ', (' year ')'}
Result := E transduced by G'
```

partitioned by, defined on P-STRING, performs an intra-p-string comparison with the argument FUNCTION (returning a value `F` for each node) that is applicable to all subtrees of the original p-string `P`. The operator returns a p-string that groups the subtrees of `P` by their `F` values. The result is a set of PARTITIONS. **where**, defined on P-STRING, takes a boolean function and returns the P-STRING with all subtrees removed where the application of the boolean function fails.

5. PAT

Introduction. The PAT text searching system [GONNE87, 91] has been developed at the University of Waterloo, *Center for the New Oxford English Dictionary and Text Research*, where it is used to query the Oxford English Dictionary (OED). The PAT text model is implemented as a text retrieval engine that communicates with the user either directly by PAT expressions, or by a tailored windowing interface called LECTOR [SALMI92].

Objects. PAT is founded on indexed text, where the index terms are specified by a grammar: the so called *text installer* builds up these indexes by defining several term patterns, that each result in a separate index.

As the grammar is not context sensitive, some problems may occur in defining indexing terms, e.g. 'word' would be defined as a sequence of characters between interpunction, which will distinguish the 'words' `U`, `S`, `policy`, `making`, `ice`, `cream` in '`U.S. policy-making on ice cream`'. Moreover, SGML encoded documents will cause problems when the slash (/) is defined as interpunction, due to the end tag specification in this language. There's no model intrinsic reason why a context sensitive grammar has not been defined (yet). The designers state somewhat evasively that *"the richness of natural language and the richness of information needs from natural language texts will always cause problems in defining indexing satisfactorily"* ([SALMI92:8]).

Within the same index the elements may not overlap and are always of the same nature, e.g. characters (proofreading), words (text retrieval), structural sections (hierarchical views). Each element starts a *semi-infinite string*, i.e. the sequence of characters starting from the element upto the end of the text. Searching is based on the match between query and index term: if the indexed elements are defined as 'character', the text is fully accessible. An advantage of treating texts as build out of such semi-infinite strings is that no superimposed (precompiled) text representation is required.

The evaluation result of expressions in PAT are result sets that consist either of match points or regions. Therefore, we may distinguish, among others, the following datatypes. CHARACTER is the smallest indexable element. STRING is defined as any sequence of characters. Strings are always normalized (both as subject and object), i.e. interpunction characters are converted to blanks. This restricts searching for terms with specific delimiters. MATCH-POINT indicates the start of a semi-infinite string. Each match point has a unique position in the text. REGION is a text section between two characters. REGIONS always belong to one or more region sets (non overlapping). Typical examples are `year` (a sequence of 4 digits between interpunction), and `paragraph` (all text between `<paragraph` and `</paragraph`, where '>' is treated as interpunction). The support of regions in this sense distinguishes the PAT model from conventional full text retrieval applications, though not from other text models (cf.

containment model, MdF, treated in sections 7 and 8 respectively). MATCHPOINTSET is an (unordered) result set containing MATCHPOINTS. REGIONSET (also a result set) is an unordered set containing REGIONS.

Operations. PAT expressions are defined over these basic datatypes, and each one may serve as subject and object in any expression. Below the operations defined are described.

First, the range operator (**..**) is defined over STRINGS. The operator implements a generator over its elements, each element lexicographically between subject and object, inclusive. For example, `"mi".."mo"` is evaluated as a set of strings {`"mi", "mj", "mk", "ml", "mn", "mo"`}. Next, a lexical search will locate the argument in the text via the index. It returns a MATCHPOINTSET containing all matchpoints for the argument. If the argument is a single term (as in line 1 below) all positions of that term are collected; if a *range* of terms is specified (as in line 2), all terms lexically between the prior and the latter are collected:

```
1     "You will stay there anyway" "ay"
2     "You will stay there anyway" "ya".."yz"
```

The 'cursor' in the subject string may be moved directly. In that case, a MATCHPOINTSET of all match points that are at a specified position is returned. For example, absolute positions are denoted by **[***p***]**. The following will return one match point.

```
"You will stay there anyway" [10]
```

As another example, the **shift** operator moves the cursor a number of positions relative to the current cursor position, returning a MATCHPOINTSET. The expression below shifts 2 positions to the right.

```
"You will stay there anyway" shift.2 "w"
```

Frequency search operators deal with the frequency of terms in the subject, and always return a MATCHPOINTSET. **signif** is defined in two ways. First, the *most frequent* substrings consisting of whole indexed elements may be retrieved, beginning a phrase starting with the given object string. For instance, `signif "a"` will (in case of a word based index) return the set of match points of the most frequent words in the subject string starting with 'a'. **signif** may accept a second parameter (in that case a dot indicates a parameter passed) that specifies the number of consecutive indexed elements that should be compared. Second, if a negated parameter is passed, the **signif.-***n* will return the *n* most frequent elements in a *sequence* of match point sets. For example, `signif.-3` will return the match point sets for the three most frequent terms in the source string.

The **Lrep** operator is similar to regular indexed retrieval, but is designed to find the *longest repeated* substring. It accepts a match point set as its argument, and returns a subset of the argument set that identifies at least two longest matching extensions. If an integer argument is passed (as in `lrep.6 'a'`), of all semi-infinite strings that are retrieved by a regular **lrep** operation, the results that are closest to the integer argument specified are returned.

The **docs** operator returns a REGIONSET that starts and ends with any character of the MATCHPOINTSETs entered as an argument. For instance, the following expression would return region set with one element, identifying '`ay there any`':

```
"You will stay there anyway"
              docs ("ay")..("ya".."yz")
```

Several operators work on sets and return a subset — the type of the subject determines the type of the result set. **including** returns the subset that contains the argument string. If a text is defined as a set of regions denoting sentences (symbolically referred to as `*S`), `*S including "of"` would return the subset of sentences containing the string '`of`'. PAT also implements regular set manipulation operators (for difference, union, intersection). Finally, the **fby**.*n* and the **near.***n* operators are used when the members of the subject set must precede some member of the object set by at most n characters (implementing adjacency). The **within** operator returns all regions in subject that are contained in the object set, e.g. `sonnetlines = *line within *sonnet`.

6. TOMS

Introduction. The TOMS model (*textual object management system*) is developed and used as part of a full text retrieval system. It is a primary indexing toolkit, i.e. a series of algorithms that create an index surrogate for superimposed text structures, i.e. all hierarchical text units beyond the word token.

TOMS deals with textual objects — those components of texts that are of interest and that may be recognized and typed. The management of these objects includes the description of structural relationships, their recognition and access in the textual object.

First, TOMS is designed to *identify textual objects* in a given set of texts. This may concern any sequence of alphanumerical characters, such as words, paragraphs, chapters. Each individual object is assigned a type. A provision is made for the assignment of attributes. Second, TOMS defines and types the *relationships* between these textual objects through document grammars, and makes this information persistent. The relations are strictly hierarchical, however, the designers are planning to incorporate more elaborate kinds of relations such as reference and recursion. Third, TOMS is based on the definition of *recognizer functions* that deal with object recognition. Thus the grammar activates a set of recognizer functions stored in a library, each of which is responsible for the recognition of a typed object instance.

Recognizer functions are bound to a class of textual objects (see example below). They accept a string of characters in a document, and return a list of *(offset, length)* pairs of all the objects of their class found within that text. The combination of recognizer functions and the document grammar make up a *document parser*. When the individual objects (tokens) are recognized by the parser they are collected in a *primary index* (also concordance list). All tokens are indexed by their type in a *secondary index*. The primary index is central to the TOMS approach, i.e. any recognizable (and typed) sequence of characters in the text will end up in the index.

In TOMS, structuring constructs, based on —but not conforming to— the ODA convention [ISO8613], describe the possible relationships between the textual objects. These are:

REP	Repetition of similar objects (shortened by the use of square brackets).
CHO	Alternative objects.
SEQ	Sequence of objects.
PAR	Parallel objects.

For example, the following grammar describes an e-mail message ([DEERW92:132]):

```
message SEQ (
        status,
        headers [
                CHO (
                adhead SEQ (
                       label,
```

```
                    [address]),
          nahead SEQ (
                    label,
                    value [valword]))],
     body [sentence [word]])
```

Object class references are printed in bold. They are recognized by associated recognizer functions.

Objects. This subsection describes the kinds of objects used by the TOMS system. A description of operations is not included, as they are not clearly described in the source reference.

An INDEX is the combination of a primary and secondary index. The index is used to access individual objects (primary) of any type (secondary). The document structure (available as a MARKING) is separated from, but navigationally related to the index. A MARKING is a structure tree created when a grammar is applied to the text, much like a P-STRING (see section 4). It is immutable — therefore, no structure recompilation operations are defined. Markings may be traversed and interrogated by the user of typed CURSORs. Each word and each higher level object has its own cursor. Regular tree navigation is supported, always returning a cursor. In addition, the function **context** accepts an object and a cursor and returns a list of textual objects that are contained in, or contain the object provided as an argument. The cursor determines the type of the object returned: if the cursor is of type word, all words are returned in a list. The **project** function returns the first cursor that is a decedent of the given cursor and which matches the given label (type).

A CURSOR holds information on the current object in the marking. Each object may have its own cursor. Each cursor has a 'type' (i.e. is associated with a recognizer function). Finally, an OBJECT-LIST holds the result of the **context** operation offered by MARKING. This implements a **getobject** operation that returns each consecutive object in the list.

7. The containment model

Introduction. [BURKO91, 92a, 92b] describe the *containment model*. This model provides operations and data structures for a text-dominated database with a hierarchical structure. It tries to bind the features of data retrieval and information retrieval systems, in particular the ranking strategies applied to word collections in text retrieval environments. The model is implemented as a *retrieval engine*; the Textriever system that interfaces to the engine is described in [BURKO91]. Each local text hierarchy within a document, e.g. each chapter, is recorded in a *tag and hierarchy specification* (THS), which keeps track of tags valid for that hierarchy, the structure of the hierarchy, and interface menu items that apply for that type of hierarchy. The texts are stored in a *database*, which is partitioned into *data collections*. Each data collection starts with a THS file specification (somewhat similar to an SGML DTD), followed by the document collection title and a representation of various local hierarchies.

In-line encoded extents (cf. SGML *exceptions*) are separated from the 'main' hierarchical structure (also called the *spinal sequence*, cf. SGML *elements*), but may be used for specification or cross-reference purposes.

Objects. The containment model is based on *contiguous extents*, much like 'regions' in PAT defined as linear character orderings between start- and end positions. If these extents do not cross, they are *disjoint*. If one or more (disjoint) contiguous extents together have particular significance in the database, their union is called a *text element* (cf. SGML *element*).

A *concordance list* denotes the list of all text elements that are of the same type, such as all words, all chapters, and so on. All concordance lists together defined on one text or text corpus denote the structure of the text, and allow for structured retrieval through *retrieval command strings* (RCS). These commands are expressions that activate *filters* of two types: selection and rejection.

Anticipating the subsection on operations, some examples of a RCS are given below (evaluated from left to right). The first RCS will select all chapter titles within chapters that contain the word `highlands` or `mountain`. The second RCS however locates `chapter_titles` that contain both words:

```
result1 = <chapter> SW {<chapter_title>
          SW {"highlands", "mountain"}}
result2 = <chapter> SW {<chapter_title>}
          SW {"highlands"} SW {"mountain"}
```

A DC-EXTENT (*disjoint contiguous extent*) is the representation of a textual item. It holds the start position and end position of the series of text symbols. Contiguous extents may be static (created when the text is added to the database) or dynamic (defined at retrieval time). A CONCORDANCE-LIST is a collection of disjoint contiguous extents. Concordance lists may be symbolized and used in subsequent expressions. For instance, `<chapter>` denotes all concordance lists that are labeled as *chapter*.

A RESULT-LIST is built for each expression, but remains within the retrieval engine and is transient. It is identified similar to retrieval sets in more conventional information retrieval environments.

Operations. The following operations are defined for the containment model.

CONCORDANCE-LISTs implement four operations. *Select narrow* (**SN**) is interpreted as 'is contained in' i.e. selects the concordance lists that are contained in the concordance list passed as an argument. One may thus select the concordance lists 'of lower level'. For example, `<chapter> SN <paragraph>` selects all paragraphs that are part of a chapter. *Select wider* (**SW**) is interpreted as 'contains', and returns a concordance list of all extents that contain the argument: `<chapter> SW {"Information retrieval"}` returns all extents labeled as *chapter* that contain the phrase given.

Reject narrow (**RN**) is the opposite of **SN**, while *reject wider* (**RW**) is the opposite of **SW**.

RESULT-LISTs implement five operations. `var_name[`*n*`]` returns the extent at index *n* in the result list. `|var_name|` (*cardinal*) returns the number of extents in the result list. `var_name(`*n* **:** *m*`)` returns the sub-result list of the given result list that runs from position *n* upto *m* (including). **length** returns the *word length* of the extent passed as an argument. For instance, `length{result_list(2)}` will return the word length of the extent at index 2.

A typical derived operation is **rank**, based on the IDF (inverse document frequency) formula of [HARMA90]. It accepts a previously compiled result list, a label, and a sequence of terms that are treated as the query terms:

```
ranking = RANK{list_of_docs, <doc>,
               "mountain", "highland"}
```

The operation returns a result list that contains all extents that are presumed to be most relevant for the user's information needs (put down in the term list).

8. MdF

Text models define both the structure of the textual sources and how they are manipulated in terms of objects, operations and constraints. [DOEDE94] only focuses on text representation. This is partly due to the complexity of text handling. All text models are founded on the assumption that the texts are well formed with respect to consecutiveness and grammatical (mostly hierarchical) structure. This is realistic in controlled situations, eg. checked corpora, validated dictionaries, and the like. However, problems occur when texts function in less controlled or more complex situations, as in corpus linguistics, where linguistic structure is the focus of text interrogation and relations between linguistic objects may be of any type, and in text criticism, where texts are incomplete or corrupted.

The MdF model (Monads-dot-Features) has emerged from the work on the ECA linguistic database (*Electronic Concordance Application*). MdF is based on the definition of textual objects and the relations between these objects. These objects represent a set of (possibly non-consecutive) *monads*.

Objects. The two main types in the MdF model are MONAD and OBJECT. A MONAD is an absolute, indivisible position in a text, which corresponds to e.g. a word. Each monad is assigned a unique number, i.e. the first monad recognized in the text string is numbered as 1, the second as 2, and so on. Monads are grouped as OBJECTs, e.g. words are grouped into sentence objects, sentence objects are grouped into paragraph objects etc. They thus form typed sets of monads. The type of an object assigned to a group of monads defines (as always) the number and kind of *features* (attributes) that are valid for that group (eg. `text`, `surface` and `part_of_speech` for `word` objects). Such features are used to convey information on the object, and includes the text that is associated with the monad (e.g. the word form itself). Features may be grouped as (un)ordered sets.

Objects are also numbered (*ordinal*): the combination of object type and its ordinal is called the *object name* (the second word in a text is `word-2`). The name uniquely identifies each object.

As stated, features may be used to denote facts (such as categoric information) and relations (links with composed clauses). No restrictions apply to these features: their values may even have an internal structure. This immediately implies that no restrictions are imposed on the user as to how relate objects. For instance, there's no scheme or grammar to go with the model that enforces constraints and therefore preserves integrity on the structure level.

Operations. The model as described is simple, because it is very abstract. There are no immediate constraints on the way the model is used to describe the textual source. MdF does not define any way of accessing and manipulating the MdF conforming database. This has some advantages, that will be outlined here.

As monads form (ordered) sets of relevant locations in the text, the *part_of* relation between object$_1$ and object$_2$ may be defined as a subset relation over monads collected by object$_1$ and object$_2$. Most hierarchical relations may be described by the part_of relation.

The MdF model allows the **part_of** relation to be abstracted into objects. **part_of** is an ad-hoc notion: it may turn out that a specific set of monads is part of a (equally large, or larger) set of monads. This does not mean that type related assumptions on this containment exist. If these would exist, the part_of relation may be formulated by two notions that in fact describe composition: *covered_by* and *buildable_from*. **covered_by** denotes the **part_of** relation; **buildable_from** denotes the **has_parts** relation. These notions are type specific.

There are circumstances where ranges of text positions may overlap. This typically occurs when two concurrent object schema's exist on the same text. The same would hold for a text grammar that deals with page layout, and one that deals with content structure. Overlapping sets are supported by the MdF model.

Normally, monads represent consecutive text elements by positions, and objects collect and type these sequences accordingly. These text elements are represented in a feature. However, ranges of monads may be disregarded. If the monads of an object are not consecutive the object is said to have one or more *holes*.

Finally, as objects may have 'holes', the notion of *consecutiveness* is considered. Linearly ordered monads, however derived from the original text, are called a *universe*. For example, `"John, called Mary."` is a universe `U` of `"John, who had missed the last train, called Mary."`. Consecutiveness is defined over substrata, i.e. `"John,"` precedes `"called"` with respect to the universe `U`. Such a concept may be valuable for structured documents that contain tags that should be disregarded in certain contexts (e.g. linear searches on the running text).

9. The Bayan system

Introduction. According to [KING90:12], *"the primary function of text management systems is to provide for the storage and manipulation of documents. An additional function of these systems is to provide a means for conveying the meaning of the texts that they manage."*

The authors describe an object-based system designed to handle Arabic texts in particular. According to the authors, a new system design was needed because current models had some shortcomings: their data abstraction capacities were too limited, and too little attention was given to problems concerning graphemic word- and sentence composition in the Arabic languages. The former was resolved by implementing words, texts and documents as objects, as will be treated below. The latter was resolved by contextual analysis of word fragments in Arabic: *"Instead of simply trying to adopt an environment to Arabic, the properties of the Arabic language were the starting point and everything was designed to meet the needs of Arabic, thus avoiding the shortcomings of other projects."*

Objects and operations. The Bayan system supports three main object types. These will be mentioned here; their (minimal) operations will be explained below each object specification. The basic object identification is not specified, but are collected in lists (typed as OBJECT_ID) to represent sequences of text units. Although several simple types are supported (DATE, AUTHOR_ID, ACCESS, STATUS), and at least an additional LIST class is supported to collect TEXT_UNITs, these will not be treated further.

An ARABIC_DOCUMENT contains the elements that make up the document and some additional features. These elements may be other ARABIC_DOCUMENTs or TEXT_UNIT objects. This object class is the only one that implements composition. The operations rendered by this ADT are given here.

The **list_all_elements** operation returns a list of TEXT_UNIT objects that the object is composed of. This listing is recursive, such that collected elements of type ARABIC_DOCUMENT will list their elements in turn. **output_text_body** actually prints or displays the result of **list_all_elements**. **add_an_element** requires a position and an OBJECT_ID, and inserts the objects at that location in the document. The operation is not specified further; the concept of 'document position' is absent in [KING90]. The same holds for **delete_element**, that requires the same parameter types.

A TEXT_UNIT holds the indivisible text parts that make up an ARABIC_DOCUMENT. The text is input directly into the object using the **edit** operation; no reference to a grammar is given, nor is such a reference intended. The operations rendered by this ADT are **output_text_body**, which outputs the body of text in the object (which is of type ARABIC_TEXT), and **edit**, which invokes an Arabic text editor on the text body. Objects of type ARABIC_WORD contain the linguistic root of the Arabic word, (root-to-wordtype) derivation rules, and categoric information. The operation **add_rule** appends a derivation rule to the ARABIC_WORD instance. A specific rule may be applied to a root using the **make_derived_word** operation. The applicability of the rule is tested by **is_legal**. **match** tests if the word given matches the root word or any derived word type. Finally, all possibly derived words are listed by the **list_all_derived** operation, i.e. all derivation rules are applied to the root.

The operations rendered by all objects, apart from the operations listed here, deal with memory management (and are therefore not relevant to the survey). Apart from these, **create_object** accepts an OBJECT_ID, and creates an instance of the class that implements the **create_object** operation. The operation **modify_attribute** accepts an OBJECT_ID, an attribute name and a value, and will update that attribute accordingly. **delete_object** will remove the object identified by OBJECT_ID from memory and the database.

Bayan is designed to implement a TBMS for Arabic texts. Therefore a lot of attention is given to the environment in which the objects will function.

The objects are located in the *object manager*, which works like a server with a restricted interface. This object manager holds the *database* and object *buffer* mentioned above. It also holds information on the classes and the general class methods. The object manager *I/O interface* uses drivers to type and display Arabic text, using the contextual analysis module.

The *query manager* receives a word and returns a list of object identifiers (of documents or text units) for these words or derivations of these words in all documents. The search is based on indexing, such that words are related to documents and text units by special pointer structures (not by object identifier).

The *text manager* is responsible for document input and output. The text converted to internal object format (ARABIC_DOCUMENT, TEXT_UNIT) is passed to the *presentation manager*. This module interacts with the application, or displays the information on the screen or printer.

10. Other models

Some models have been presented that only give an overview of the structure of textual objects. These models do present a grammar for document parsing, but provide no information on access languages or object behavior. Therefore, these 'models' are gathered here.

'Extended MAESTRO'. The model described by [BARNA91] focuses on hierarchical relations between *document collections*. These collections may contain both documents and other document collections. Documents are typed and may be attributed. The document attributes may be of two types: TEXT and REFERENCE. Thus, in Maestro a document `CompJType` may be defined as

```
document: CompJType is PaperType with Date text
```

`PaperType` is a named document grammar, `Date` is the attribute of the document, and `CompJType` is the name of the binding between the grammar and the attribute. **document** is a declaration keyword. The grammar in Maestro takes the form of a rudimentary SGML DTD which lacks attributes, minimization and exceptions, and cannot handle concurrent document markup.

The model uses *zones* to store the location of all element contents in the running text. Each new text element found is assigned an OID, a link to a parent element, and a zone. This produces a tree structure, of which the terminal elements point to a zone. This requires the text to be immutable, and access to the zones is based on hierarchy traversals.

Grif. The text model of the Grif system [QUINT89] is based on the use of three languages S, P (presentation language) and T (exchange language). In S one declares the hierarchical and sequential structure of a document:

```
STRUCTURE Book;
    Book=BEGIN
               Title=TEXT;
               Author=TEXT;
               Body=LIST OF (chapter);
    END;
    Chapter=BEGIN
    ...
```

The TEXT type is a character string; the LIST indicates a sequence structures of the paramerized type. Each root object (`Book`, `Chapter` etc.) defines an object that may be used in aggregations and references. A document may be created using any root object description (e.g. `Chapter`). Thus documents may be built in a modular fashion. References to other text objects are defined by a special language construct **UNITS**. Constraints on document manipulations are thus regulated by the **STRUCTURE** description. Attributes of text objects are supported.

Multos. [LUTZ89] assumes that a hierarchical, type-oriented document model may be defined. In this, the ODA standard is followed.

The grammar used to identify the document structure is called the *conceptual structure definition* (CSD). This records three main principles. First, documents are tree structured (aggregation of low level objects). This structural knowledge extends classical retrieval by allowing logical sections of documents to be specified. Second, each conceptual structure is typed. The type of the object is defined by its structure (as abstracted in the grammar). Third, the type system is extended by refining an 'open' component in the supertype definition (the 'spring' component).

Documents are preprocessed to convey layout and logical knowledge. The recognition process (reading documents, mapping their form to the conceptual structure) is controlled by the *content description language* (CDL). This language defines the rules for classification (typing), by supporting all kinds of recognition predicates, such as layout and logical forms, pattern matching, structural knowledge (linked with the CSD results), references, etc. The result of this preprocessing is a content representation, that may be used in retrieval. [LUTZ89] reports on some findings in terms of relevance.

11. Discussion

In this concluding section the models are compared on the basis of the three *model evaluation criteria* established in the beginning of this paper. These criteria are structure, operations and constraints defined on the logical level. Some general *restrictions* of the models are described, as these restrictions point out what augmentations are expected in future work in this area. Future text models or applications should take advantage of the time and effort invested in the models outlined here. The descriptions given in the preceding sections outline some of the merits of the models. In this last section I also refer to the text characteristics as mentioned in section 2.

Logical structure. All models support text access based on *linear order* and *containment* relations, that take the form of hierarchies described by *grammars* (p-string, TOMS, extended Maestro, Multos, Grif, TDM to a certain extent), or are recognized ad-hoc using *set-based operations* (Containment, TDM, MdF). Grammars are usually context free; only one model incorporates *conditional recognition* (p-string).

Although grammars are central to text structuring, they are limited in extent. A context-free grammar would not be able to reject a sequence of footnote references without intermediate running text. No general context (in)sensitive rewriting grammar can describe the occurrence of in-line elements such as highlighted phrases, annotation marks, and so on — at least not in an elegant way. Only the p-string grammar format allows for some dynamic conditions in recognition that restricts the scope of the rewriting process. However, even this grammar is too straightforward to recognize and abstract all (realistic) textual objects. The Containment model recognizes in-line elements, for example cross-references, but does not allow for constraints on this level.

The PAT model does not recognize the concept of hierarchies. It focuses on linear sequences that may be typed and indexed arbitrarily. The grammar is used only to recognize these objects. Nested relations are not maintained when scanning the source text. Furthermore, all semantics attached to specific strings are recognized at retrieval time. The exclusion of a cross-reference from the source is a retrieval operation (set difference). Bayan uses no grammar but links sections into a tree of objects. Bayan may be considered as a structured text editor, augmented with linguistic knowledge representations.

Some models use (part of) a *standard on text encoding* (ODA: TOMS, Multos, SGML: extended Maestro) to recognize and interpret structural aspects of the texts. SGML is usually only interpreted as a code format, not as a structuring strategy. None of the models will correctly process real-life *SGML encoded documents* (unless in controlled circumstances, PAT). For instance, the Containment model assumes the source to be tagged in a transparent way. The markup language advocated is SGML. However, the inclusion of SGML DTDs and declarations, or SGML attributes is not modelled. SGML is thus used as a tag list only, that is reflected in the local THS grammar. This grammar is not elaborated upon in the references.

Relations put down in the source text are usually not recognized and are recorded externally if supported at all (MdF, TDM). For a hypergraph based PAT extension, see [RAYMO88]. An exception may be the CDL reference recognition of Multos, though the rationale of this particular recognition process is unclear.

Multiple structures are modelled only partly (TOMS, MdF), and none of the models go into the intrinsic relation between such parallel structures.

Although more than one grammar may be defined on the same source (thus implementing concurrent text grammars) it remains unclear how the relations between these parsed structures are modelled by the p-string model. The model does not define operators that link different elements in two or more concurrent p-strings on the same text sequence, allowing for queries such as "what kind of substructure starts at page 14?" or "Where do sentences pass the line boundaries?".

In TOMS, the inclusion of the **PAR** operator allows for multiple 'views' on the same document. However, it is not made clear how the same document may be recognized by two or more parallel encodings. It seems reasonable to assume that the recognizer functions know about embedded tags (or more general: patterns) that do not belong to the current element, and that should be skipped.

The textual objects are interpreted as sections — *regions or cursors* (PAT, TOMS, Containment, Maestro) and/or *offsets* (semi-infinite strings in PAT) — that are related by containment or succession, or are *strings* in their own right (p-string, MdF, TDM).

The separation of *character functions* is made by none of the models defined here. It is unclear how intermediate codes or characters are filtered out or replaced by other sequences when viewing the text. None of the models explicitly mention multiple character encoding or replaceable character sets. The models focus on 'clean', though possibly tagged texts.

Attributes are supported to a limited extent, and are never extracted from the source text. Attributes are limited in definition and type (Maestro, Grif). The MdF model, though clearly based on attribute definition, does not restrict or regulate the use of the object features. It therefore does not allow any structuring strategy to be formulated ('anything goes'). TOMS also allows for the definition of object-related data attributes. However, these are not part of the text, and cannot be recognized as such.

Logical operations. Most of the models focus on the expressions used to access the text. These expressions generate *sets* (TDM, PAT, Containment) or other datatypes such as *parsed string representations* (p-string). In this respect, the p-string model is transparent and relatively flexible. However, the source as well as the grammar are immutable objects — changing the source will have to result in a complete reparse (an alternative is treated in [GYSSE89])

The TOMS model described focuses on a tree traversal strategy (MARKING). This differs from the other models that use set manipulation or string scanning to query the structural parts of the text. Linear order in text structures is recognized.

The operator set is sometimes defined on a *minimal* basis (Containment, Bayan), sometimes more *complete* (PAT, p-string). Operations usually concern the *retrieval* of items from an immutable text (see below) and *specialized functions* (statistical: PAT, document ranking: Containment, linguistic: Bayan). Operators are sometimes combined to define *text views* (TDM).

None but the Containment model allow for the definition of *operations* that are confined to specific text structures. Thus all operators apply to all textual objects: a subject line has the same functionality as the body of a letter. In the TDM model the behavior of a paragraph is identical to that of a chapter, although its structure may be different.

As stated, the models do not, or to a limited extent, support source *updates*. The p-string model does allow a reparse of already parsed strings. However, this is done in a 'batch' fashion: the entire string is reparsed on command. Edits to parts of the string (that may violate its integrity) are checked by explicit invocation of the parser. As far as I can see, the operations defined by TDM are transient and are only used to extract formatted information from a fixed source.

Logical constraints. Integrity through structural or operational constraints is not described explicitly by any of the models. However, structural integrity enforcement through grammar definition is a natural part of all grammar-based models (in PAT, constraints on text sectioning are defined as part of the indexing process — this is the only kind of integrity enforcement available). Conditional rewriting (which could probably be used to define a context sensitive parser) is part of the p-string model. This model only provides for the definition of operators that apply to parsed strings. No provision is made for constraints on the elements recognized while parsing the string. However, [TAGUE91] and [SALMI92] extend the p-string grammar by defining *constrained schema's*, i.e. grammars that replace the right hand side of the production by a set of properties that are treated as constraints on the production.

In my view, integrity is part of a *textual object type* and the appropriate rules should be actualized by the instances of the type (including its possible structure). If (as in most of the models) the textual objects are not abstracted it will be very cumbersome to augment them with such constraints.

Note. *This article is an extended abstract of a chapter of the PHD thesis of the author. The thesis entails the definition of a textbase management system (TBMS) for humanities text-based research. The system proposed models structural aspects of textual data, and conforms to SGML. The more fundamental modularization strategy is object-oriented. Neither SGML nor the TBMS itself will be discussed in this article. SGML does not model behavioral aspects of texts, though the language highlights some very important textual features mentioned in this article. The TBMS proposal is still under development. The thesis is due November 1994. The complete chapter on text models is open for public comment.*

12. References

BARNA91 D. T. Barnard et al: 'SGML documents and non-linear text retrieval'. In: [LICHN91] pp 226-244

BURKO91 F. J. Burkowsky: 'The use of retrieval filters to localize information in a hierarchically tagged text-dominated database'. In: [LICHN91] pp 264-284

BURKO92a F. J. Burkowski: 'Retrieval activities in a database consisting of heterogeneous collections of structured text'. In: [SIGIR-5] 1992, pp. 112-125.

BURKO92b F. J. Burkowski: 'An algebra for hierarchically organized text-dominated databases'. In: **Information processing & management.** (Oxford) 1992

DEERW92 S. Deerwester et al: 'A textual object management system'. In: [SIGIR-5] pp. 126-139.

DESAI86 B. C. Desai, P. Goyal, S. Sadri: 'A data model for use with formatted and textual data'. In: [JASIS] 37 (3) 1986 Pg. 158-165

DOEDE94 C. J. Doedens: **Natural and formal language access to text databases**. PhD Thesis, Univ. of Utrecht, Holland, 1994 (to appear).

GONNE87 G.. H. Gonnet, F. W. Tompa: 'Mind your grammar: a new approach to modelling text'. In: [VLDB] 13 (1987). Brighton, 1987. Pg. 339-346.

GONNE91 G. H. Gonnet et al: 'Lexicological indices for text: inverted files vs. PAT trees'. Technical report OED-91-01, University of Waterloo, 1991

GYSSE89 M Gyssens, J. Paredaens, D. vd. Gucht: 'A grammar based approach toward unifying hierarchical data models' Extended abstract. In: [SIGMOD] 1989. Pp. 263-272

HARMA90 D. Harman, G. Candela: 'Retrieving records from a gigabyte of text on a minicomputer using statistical ranking'. In: [JASIS] 41 (8) 1990 Pg. 581-589

ISO8613 **Information processing - text and office systems - office document architecture (ODA) and interchange format.** International organization for standardization. 1989

ISO8879 **Information processing - Text and office systems - Standard generalized markup language (SGML).** International organization for standardization. 1986, with amendment 1, 1988 **(ISO 8879-1986/A1:1988 (E)).**

JASIS **Journal of the American society for information science.** Washington D.C. (ASIS)

KING90 R. King, A. Morfeq: 'Bayan: an Arabic database management system'. In: [SIGMOD] 1990. Pp. 12-23.

LICHN91 **Intelligent text and image handling. Proceedings of a conference on intelligent text and image handling 'Raio 91', Barcelona, Spain, 2-5 April 1991** Edited by A. Lichnerowicz. Amsterdam et al, 1991 (Elsevier)

LUTZ89 E. Lutz: 'Knowledge based classification of office documents'. In: [WOODM89] pp 353-362

QUINT89 V. Quint, I. Vatton: 'Modularity in structured documents'. In: [WOODM89] pp 170-177

RAYMO88 D. Raymond, F. Tompa: 'Hypertext and the Oxford English dictionary'. In: [CACM] 31 (7) 1988. Pp. 871-879

SALMI92 A. Salminen, F. W. Tompa: 'PAT expressions: an algebra for text search'. Papers in Computational Lexicography: COMPLEX '92, Proc. 2nd Int. Conf. on Computational Lexicography (F. Kiefer, G. Kiss, J. Pajzs, ed.), Linguistics Inst., Hungarian Academy of Science, Budapest (October 1992), 309-332. (also available as Technical Report OED-92-02).

SIGIR-N **Proceedings of the** N**th annual international ACM SIGIR conference on research and development in information retrieval.**

TAGUE91 J. Tague et al: 'Complete formal model for information retrieval systems'. In: [SIGMOD] 1991. Pp. 14-20

WOODM89 **Woodman '89. Workshop on object-oriented document manipulation. Rennes, France, 28-31 may, 1989.** Preprints, edited by J. André, J. Bézivin. In the Bigre/Globule series, nr. 63-64, May 1989. (Afcet, Bigre, Ccett)