Technical Briefing

Information Retrieval Techniques for Hypertext in the Semi-Structured Toolkit

Gary Perlman
Computer and Information Science
Ohio State University
2036 Neil Avenue
Columbus, OH 43210-1277 USA
perlman@cis.ohio-state.edu

ABSTRACT

The Semi-Structured Toolkit (SST) is a C library that provides universal functions based on abstractions for storage format- and data type-dependencies of semi-structured/frame-based information units. The SST provides searching, sorting, viewing, and linking operations for data stored in its native formats, without requiring proprietary formats or conversion. Hypertext capabilities such as linking and outlining are implemented in the SST with inverted indices for each of the fields in semi-structured records. This paper describes the implementation of hypertext capabilities in the SST.

KEYWORDS

Information retrieval, Underlying technologies, Hypertext, Indexing, Linking, Outlining, Open systems.

INTRODUCTION

In this paper I discuss some ways that indexes can be used to allow implicit links to be followed in large information spaces. The use of indexes described here shows how the information in semi-structured information can be exploited with minimal, if any, authoring effort.

PREVIOUS APPROACHES TO OPEN/IMPLICIT HYPERTEXT

Fountain *et al* [Fount90] discuss some problems of hypermedia systems that *"act as possible barriers to the creation, extension and integration of hypertext document systems."* They note the **authoring effort** required to write or convert to hypertext, that hypertext systems are **closed** and do not work with other software, that **proprietary document formats** make it difficult to use data, and that **read-only media** make it impossible for users to make their own links. Systems like Apple HyperCard or OWL Guide store their links along with the data being linked in a format that is difficult to process. In contrast, the approach taken in Microcosm [Fount90], like that taken in many *open* hypertext systems (e.g., [Pearl89]), is to separate the links from the multimedia chunks of information being linked. The effort invested in creating the linked information is not lost.

A different approach is taken in systems like SuperBook [Remd87], Document Examiner [Walk87], and NaviText [Perl89] in which links (the names of link targets) are embedded in hypertext, but do not have a separate existence. Instead, the links are traversed by searching an index. SuperBook [Remd87] and NaviText [Perl89] add structure to a document with simple markup languages. Document Examiner [Walk87] have only one type of link, but supports several views for it: linked information can be **included** at the source, a **precis** can be displayed, a note that the **cross-reference** exists can be displayed, or the name of a topic can be **implicitly** inserted. All these systems simplify the creation of simple links, adding functionality to links as they are traversed. They are not, however, open systems that allow the separation of data from links.

LINKING USING EXISTING IMPLICIT LINKS

The approach taken in the Semi-Structured Toolkit (SST) hypertext capabilities is a combination of the above approaches. Data can be maintained in its native format, and existing implicit links are recognized in the text. (By

naming the source and destination of a link, links can be added, but this is not a definitive feature of the scheme.) A question immediately arises: *If only existing implicit links can be traversed, what is the use of the system?* The answer is that at minimal cost many benefits can be gained by exploiting existing structural information. Consider the example in Figure 1.

Figure 1 shows page 246 of **MIL-STD-1472C** [DOD83] a highly structured document. Each paragraph contains a hierarchically structured identifier called a *milnum* (e.g., **5.15.3.5.6**), a title (e.g., Lists), and optional text (e.g., Items in lists...). Most paragraphs are sub-paragraphs of others with which their identifiers share a common prefix. Within the optional text, there might be internal cross-references to related paragraphs (e.g., **5.15.3.6.2** refers to **5.15.3.1.6**), external references to related documents (e.g., **5.15.3.7.2** refers to **MIL-STD-490**), or references to previous versions of the document (e.g., this page 246 supersedes the version of **2 May 1981**). With appropriate token recognition (hierarchical paragraph numbers and hyphenated document names are easy, dates referring to previous document versions are harder) and parsing, the reader of MIL-STD-1472C could be allowed to view and traverse the document in a variety of ways without the need to change the format of the document. Parsing a paragraph in [DOD83] is easy because U.S. Military standards follows a standard. Each paragraph can be viewed as an ordered sequence of unlabeled fields: identifier (ID), title, and text. With the knowledge that the identifiers are milnums, subordinate and superordinate paragraphs can be determined by a generate-and-test strategy; ID.N exists as an ID for another record if and only if ID has at least N subordinates. A token is an ID of a MIL-STD paragraph if it is in a specified position: indented two spaces after two newlines.

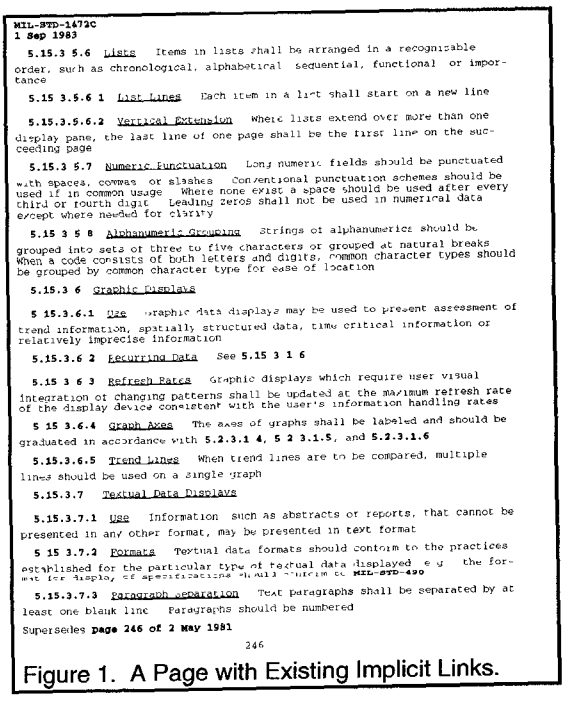
A second question is: *Are there are enough document types that are both (1) easy to parse and which (2) have a rich enough existing implicit link structure to make it worthwhile to make use of those links?* There are many other types of information with these properties [Perl92], notably electronic mail messages (with implicit links to referenced messages), electronic news articles (typically with an even greater degree of citations). A variety of special purpose systems have been created for these and other applications (e.g., calendars, bibliographic and biographic information), but the SST takes the general view of the information domains as special cases of semi-structured information (a slightly more general structure than semistructured messages [Mal87]): a sequence of named or positionally identified values, usually stored as human-readable text.

A hypertext capability designed to exploit existing implicit links may not have the expressive power of other systems, the ability to derive hypertext functionality with existing data that comes in the large quantities (e.g., mail, news, and a variety of other structured documents) recommends the development of such a capability. In addition, the low effort involved in adding links to structures precludes a hasty conclusion that the scheme can not be generalized.

REPRESENTING SEMI-STRUCTURED INFORMATION IN THE SST

The Semi-Structured Toolkit (SST) [Perl92] is a library of routines for managing semi-structured information. Semi-structured information can be viewed as a partially ordered set of fields (fields of the same name are ordered), in which fields have optional names (or are positionally-defined) and typed values. Fields in semi-structure records can identify attributes of records, or relations to other records; in that sense, they can represent nodes of information linked by typed arcs to other nodes of information. Examples of semi-structured information are mail messages [Mal87], news articles, event calendars, and bibliographic records. There are implicit links between semi-structured records mail messages and news articles refer to initiating communications, and bibliographic records can refer to their referenced publications. These implicit links create an implicit structure (e.g., directed acyclic graph, lattice, or network) that can be traversed as a hypertext.

Many hypertext operations (e.g., linking, elision) can be implemented using traditional information retrieval technologies (e.g., indexing [Frak92]) with no modification to the underlying information. Managing information in its native format is attractive because no conversion (to a



MIL-STD-1472C
1 Sep 1983

5.15.3 5.6 Lists Items in lists shall be arranged in a recognizable order, such as chronological, alphabetical sequential, functional or importance

5.15 3.5.6 1 List lines Each item in a list shall start on a new line

5.15.3.5.6.2 Vertical Extension Where lists extend over more than one display pane, the last line of one page shall be the first line on the succeeding page

5.15.3 5.7 Numeric Punctuation Long numeric fields should be punctuated with spaces, commas or slashes Conventional punctuation schemes should be used if in common usage Where none exist a space should be used after every third or fourth digit Leading zeros shall not be used in numerical data except where needed for clarity

5.15 3 5 8 Alphanumeric Grouping Strings of alphanumerics should be grouped into sets of three to five characters or grouped at natural breaks When a code consists of both letters and digits, common character types should be grouped by common character type for ease of location

5.15.3 6 Graphic Displays

5 15.3.6.1 Use Graphic data displays may be used to present assessment of trend information, spatially structured data, time critical information or relatively imprecise information

5.15.3.6 2 Recurring Data See 5.15 3 1 6

5.15 3 6 3 Refresh Rates Graphic displays which require user visual integration of changing patterns shall be updated at the maximum refresh rate of the display device consistent with the user's information handling rates

5 15 3.6.4 Graph Axes The axes of graphs shall be labeled and should be graduated in accordance with 5.2.3.1 4, 5 2 3.1.5, and 5.2.3.1.6

5.15.3.6.5 Trend Lines When trend lines are to be compared, multiple lines should be used on a single graph

5.15.3.7 Textual Data Displays

5.15.3.7.1 Use Information such as abstracts or reports, that cannot be presented in any other format, may be presented in text format

5 15 3.7.2 Formats Textual data formats should conform to the practices established for the particular type of textual data displayed e g the format for display of specifications should conform to MIL-STD-490

5.15.3.7.3 Paragraph separation Text paragraphs shall be separated by at least one blank line Paragraphs should be numbered

Supersedes page 246 of 2 May 1981

246

Figure 1. A Page with Existing Implicit Links.

proprietary format) of information is needed, and the installed base of tools can be used with the information (users do not need to give up their investment in tools and skills). Supporting linking based on implicit links is attractive because some of the benefits of hypertext can be gained without the expense of additional authoring.

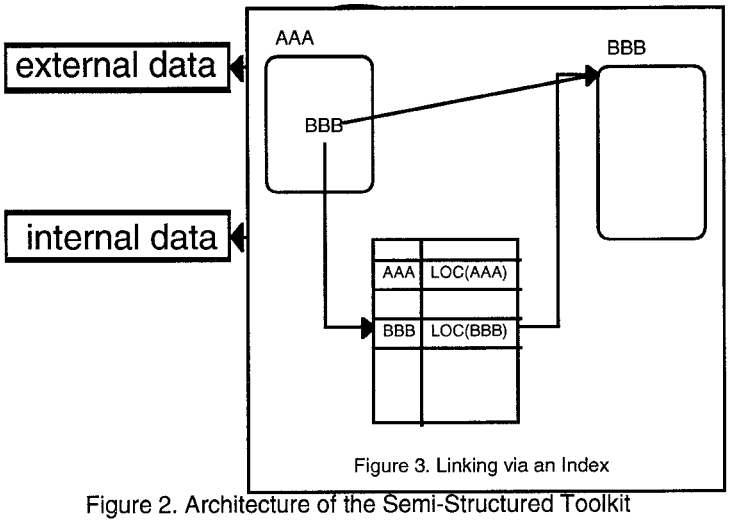


Figure 2. Architecture of the Semi-Structured Toolkit

The SST is a semi-formal system. Semi-formal systems (1) represent and automatically process certain information in formally specified ways, (2) represent and make it easy for humans to process the same or other information in ways that are not formally specified, and (3) allow the boundary between formal processing by computers and information processing by people to be easily changed [Lai88]. In effect, semi-formal systems provide support to exploit as much structure as is desired, but they do not impose structure. This is in contrast to systems such as hypertext systems or DBMS that require a commitment to a specific set of conventions that usually restrict users to specific file formats, functionality, and platforms. The philosophy behind semi-formal systems is that users can derive many benefits from minimal investment; this is desirable because the cost associated with formal systems often outweighs the benefits when applied to poorly structured problems. Semi-formal systems are most useful in information processing problems where there is some understanding of the structure of a domain, but not a complete understanding [Lai88].

The SST is designed based on a strategy of least commitment to proprietary data storage formats and application functionality [Perl92]. Figure 2 shows the basic structure of the SST and how it abstracts away the storage format and application dependence. Based on a set of lexical specifications that describe the storage format of the data, the READ and WRITE routines can operate on the data in its native format without a need for translation. The READ and WRITE routines interact with a semi-structured abstract data type (ADT) which is a virtual unbounded array of name-value fields (stored internally in a private format). All universal functions (e.g., search, sort, format) access the data through the semi-structured ADT and so do not depend on the internal or external storage format of the information; in that sense, they are universal. The functions can make use of a set of field specifications that indicate the attributes of fields (e.g., given a field name or position, field specifications indicate the data type, range, display format, etc.) Using field specifications, field values can be tokenized, checked for validity, compared for searching and sorting operations, formatted, and so on.

REPRESENTING LINKS

Links can be represented in records as follows. Assume that each record has a unique identifier (ID) stored in a known field (e.g., Message-Id or "name"). This assumption is reasonable for applications like mail and news because messages and articles are assigned a unique identifier when they are created; for other domains, such as event calendars or bibliographic entries, it is relatively easy to add an identifier field if one is not already present. A link from a point P to a record R is simply an instance of the ID for the record R at P, possibly with some markup to identify the ID of R (e.g., {ID(R)}) if the ID is difficult to distinguish from surrounding text. Such links can be qualified with offset information if needed (e.g., specific fields, and/or ranges of bytes offsets).

An inverted index of the IDs of a set of records can be built by associating the location (e.g., file name and offset) of each record with its ID. With an inverted index of the IDs for the records, hypertext operations such as linking and outlining can be implemented extremely efficiently, especially if the ID index is cached and hashed.

Figure 3 shows how a link from a point to a record can be resolved using a ID index. A point in the record on the left is the start of the link and the record on the right is the target of the link. The appearance to a user is of direct linkage from a point in AAA to BBB, but the implementation looks up the ID of BBB in the ID index and gets the paired location (LOC(BBB)). The entries in the ID index are padded to be of fixed size to speed access, but the IDs can be of any size up to some maximum determined at the time the index is built. For even large networks or records, the entire ID index can be cached (stored in main memory), and for many applications, hashing of IDs is both practical and considerably more efficient than binary search [Frak92].

REVERSING LINKS

It is possible to find the locations of all links to a particular record by searching for all instances of its ID that are not at its own location and/or not in the field holding the ID. This requires the existence of inverted indices for all fields in which the ID might appear, such as a full text index. Figure 4 shows an inverted file implementation based on Chapter 3 of [Frak92] in which the Keyword Index indicates 3 Hits at the locations indicated in the Postings file. (Although it is possible to merge the Keyword Index and the Postings file, having a uniform record size speeds processing.)

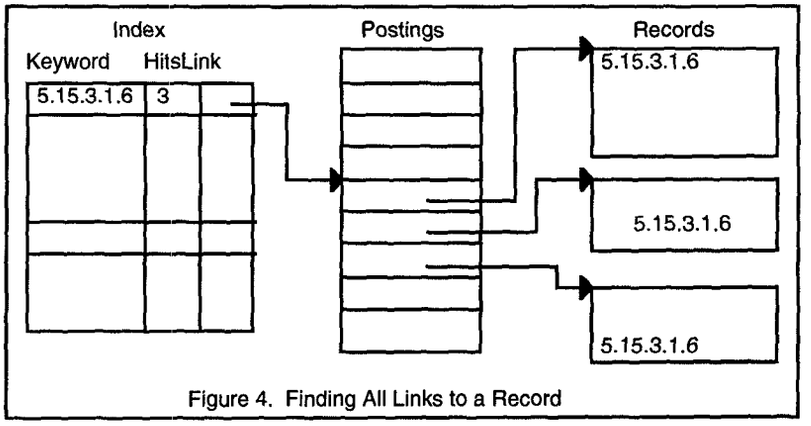


Figure 4. Finding All Links to a Record

OUTLINING

Outlining is a structural view operation that makes use of a (possibly implicit) hierarchical structure of information. An example of an outlining operation on bibliographic information would be to view the references of a bibliographic entry, then view the references of the references, and so on. First, we must assume that the field containing the references (REFS) contains the IDs of the records representing the references. (Alternatively, in the case of the document in Figure 1 in which IDs are hierarchically structured milnums, the subordinate records can be generated algorithmically.) To display the references, a formatted view of the records could be constructed with a formatted view of certain fields in each of the records (e.g., the date, first author, and title). Figure 5 shows how a ID index could be used to allow an outline view of the REFS field (the operation works for any transitive relation). The tokens in the REFS field (XX, YY, ZZ) are located in the ID index and the records at the associated locations are accessed. The outline operation makes use of the transitive property of the REFS relation and displays the contents of the REFS fields of the accessed records (the REFS of the REFS), replacing:

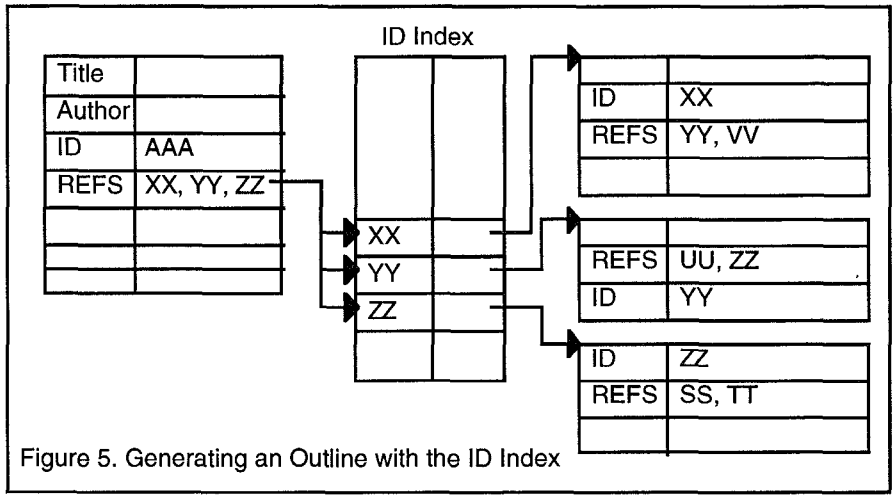


Figure 5. Generating an Outline with the ID Index

```
AAA
        XX
        YY
        ZZ
```

with:

```
AAA
        XX
                YY
                VV
        YY
                UU
                ZZ
        ZZ
                SS
                TT
```

Note that both AAA and XX refer to YY, and both AAA and YY refer to ZZ. In a textual outline form, the duplicate entries for YY and ZZ could be specially marked, or a graphical outliner could be used instead.

REVERSE-OUTLINING

Like the index-based linking operation, the index-based outline operation can be reversed. In addition to an index of all the IDs for records, an index of the REFS field (or more generally, any transitive relation field containing IDs). For example, a REFS index could be searched for XX to retrieve the locations of all the records referring to XX.

A GENERALIZED INDEX

Thus far, the linking schemes described have used specific files and/or fields to index. It is possible to generalize the linking by creating an index for each field in a file. The logical structure of the file index currently in use is shown in the Appendix. A **File** header contains information about the file, the number of different fields, the width of padding used, and other bookkeeping information. The **Fields** index contains, for each field, the name of the field, the number of terms in the field, and the address in the index where those terms are stored. The **Terms** index contains a series of keyword indexes, one for each field, with all the terms in the index, the number of records *hit* by

that term, and the location in the index where the postings are stored. Finally, the remainder of the index contains the individual postings (addresses of the records holding the terms in the fields).

In the SST, field specifications can indicate the type of the field (e.g., that a field called *author* contains a person's name, or that a field called *email* is an Internet address). With this type information, the tokenizer and indexer can determine how to recognize tokens, what text to skip, and which tokens to index (or treat as stopwords). By default, tokens are recognized as delimited by whitespace with exterior punctuation removed; this works well for many types of fields. Stopwords (e.g., *the, and, it*) are inserted in the Term index (to provide feedback during searching), but with a zero frequency and with the location holding the number of times the stopword was seen. Additionally, in the case of a single posting, the location of the posting is stored in the Term index.

In the SST, an index is built for each file so that when a file gets updated, only its index needs to be updated. Even with files up to about a megabyte, the indexing does not take long, even on low-end PCs. But the desire for efficiency for updating files creates problems, and the solution space is still being explored. To search for records in many files or to link across different documents, either multiple indexes must be searched, which is inefficient, or multi-file indexes must be built from single file indexes. The locations of records, which were offsets in single files, must be qualified with a file identifier, increasing the size of the index and the complexity of access.. When searching the HCI Bibliography [Perl93], which contains over 200 files, each with about 50 records, files are selected before they are searched (as opposed to building an index of all the files, some of which may not be of interest). An index that merges the selected file indexes can be built in the background while a query is constructed, and that index can be maintained for the duration of the session and then deleted to reclaim the space. Merging indexes is much faster than scanning and building them from scratch, and it has the advantage of allowing the flexible selection of files. This is particularly useful when indexing news articles (of which there are many thousands) which change many times a day.

The ability to select files can be views as a form of query (as can a newsgroup subscription list), and the construction of indexes of the files that are of interest serves to reduce the need for special purpose software for reading mail, news, etc. When constructing a query in the SST, it is possible to search for the same expression in a variety of fields. For example, one might look for Nielsen as an author or as an editor. To search multiple fields for the same expression, it is possible to create an index by merging the fields to be treated as equivalent (e.g., a subject index might include the title, keywords, and abstract fields). The structure of the SST File index in the appendix makes merged multi-file multi-field indexes conceivable, but they have not yet been attempted.

EXPERIENCES

We have had about four years experience using inverted indices as a substrate for hypertext operations like linking and outlining, first with NaviText [Perl89] and more recently with the SST [Perl92] applied to mail, news, bibliographic, and biographical (with digitized photographic images) browsing and querying. In addition to their known properties for searching full text databases, inverted indices are an efficient technology for implementing a variety of hypertext operations. The SST's use of implicit links in semi-structured records without the imposition of format is a new method for implementing hypertext operations. Index-based hypertext may not provide all the capabilities needed for a particular application -- there might be a need for separate existence of links if they require a richer representation -- it does provide many capabilities at virtually no cost of authoring.

REFERENCES

[DOD83] MIL-STD-1472C (1983), Human Engineering Design Criteria for Military Systems, Equipment and Facilities. Washington, DC: U.S. Department of Defense.

[Fount90] Fountain, A. M., Hall, W, Heath, I. & Davis, H. C. (1990) *MICROCOSM: An Open Model for Hypermedia with Dynamic Linking*. **Proceedings of ECHT'90**, 298-311. Cambridge University Press.

[Frak92] William B. Frakes, W. B. & Baeza-Yates, R (Eds.) (1992) **Information Retrieval: Data Structures and Algorithms**. Englewood-Cliffs, New Jersey: Prentice-Hall.

[Lai88] Lai, Y.-Y. & Malone, T. W. (1988) *Object Lens: A "Spreadsheet" for Cooperative Work.* **ACM Transactions on Office Information Systems, 6**:4, 332-353.

[Mal87] Malone, T. W., Grant, K. R., Lai, K.-Y., Rao, R., & Rosenblitt, D. (1987) *Semistructured Messages Are Surprisingly Useful for Computer-Supported Coordination*, **ACM Transactions on Office Information Systems**, **5**:2, 115-131.

[Pearl89] Pearl, A. (1989) *Sun's Link Service: A Protocol for Open Linking.* **ACM Hypertext'89 Proceedings**, 137-146.

[Perl89] Perlman, G. (1989) *Asynchronous Design/Evaluation Methods for Hypertext Technology Development*, **ACM Hypertext'89 Proceedings**, 61-81.

[Perl93] Perlman, G. (1993) *The HCI Bibliography.* **ACM SIGCHI Bulletin**, **23**:3, 36-37.

[Perl92] Perlman, G. (1992) *A Vision of Universal Functionality for Tomorrow's User Interfaces*, **Proceedings of OZCHI 92 Australian Conference on Human-Computer Interaction**, Australian Ergonomics Society, 1-14.

[Remd87] Remde, J. R., Gomez, L. M., & Landauer, T. K. (1987) *SuperBook: An Automatic Tool for Information Exploration - Hypertext?* **ACM Hypertext'87 Proceedings**, 175-188.

[Walk87] Walker, J. H. (1987) *Document Examiner: Delivery Interface for Hypertext Documents.* **ACM Hypertext'87 Proceedings**, 307-323.

ACKNOWLEDGMENTS

Chandrasekhar Reddy and Steve Edwards have been influential in recent design decisions. I would like to thank Ed Swan for his help with the X-based SST viewer, and Srinivas Raghavan for his help with the X-based SST query expression construction system.

APPENDIX: LOGICAL STRUCTURE OF THE INDEX

File Information

filename	nfields	width	...

{Fields}

field{1}	nterms{1}	&term{1}[1]
field{2}	nterms{2}	&term{2}[1]
...	...	...
field{nfields}	nterms{nfields}	&term{nfields}[1]

[Terms] in {Fields}

term{1}[1]	nrecs{1}[1]	&&rec{1}[1](1)
term{1}[2]	nrecs{1}[2]	&&rec{1}[2](1)
...	...	
term{1}[nterms{1}]	nrecs{1}[nterms{1}]	&&rec{1}[nterms{1}](1)
term{2}[1]	nrecs{2}[1]	&&rec{2}[1](1)
...	...	...
term{2}[nterms{2}]	nrecs{2}[nterms{2}]	&&rec{2}[nterms{2}](1)
...	...	...
term{nfields}[1]	nrecs{nfields}[1]	&&rec{nfields}[1](1)
...	...	...
term{nfields}[nterms{nfields}]	nrecs{nfields}[nterms{nfields}]	&&rec{nfields}[nterms{nfields}](1)

(Records) holding [Terms] in {Fields}

&rec{1}[1](1) ... &rec{1}[1](nrecs[1](1)) &rec{1}[2](1) ... &rec{1}[2](nrecs{1}[2]) ... &rec{1}[nterms{1}](1) ... &rec{1}[nterms{1}](nrecs{1}[nterms{1}])
&rec{2}[1](1) ... &rec{2}[1](nrecs{2}[1]) ... &rec{2}[nterms{2}](1) ... &rec{2}[nterms{2}](nrecs{2}[nterms{2}])
...
&rec{nfields}[1](1) ... &rec{nfields}[1](nrecs{nfields}[1]} ... &rec{nfields}[nterms{nfields}](1) ... &rec{nfields}[nterms{nfields}](nrecs{nfields}[nterms{nfields}])