Mind Your Grammar:
A New Approach to Modelling Text

Gaston H. Gonnet
Frank Wm. Tompa

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Abstract

Beginning to create the New Oxford English Dictionary database has resulted in the realization that databases for reference texts are unlike those for conventional enterprises. While the traditional approaches to database design and development are sound, the particular techniques used for commercial databases have been repeatedly found to be inappropriate for text-dominated databases, such as the New OED.

In the same way that the relational model was developed based on experiences gained from earlier database approaches, the grammar-based model presented here builds on the traditional foundations of computer science, and particularly database theory and practice. This new model uses grammars as schemas and "parsed strings" as instances. Operators on the parsed strings are defined, resulting in a "p-string algebra" that can be used for data manipulation and view definition.

The model is representation-independent and the operators are non-navigational, so that efficient implementations may be developed for unknown future hardware and operating systems. Several approaches to storage structures and efficient processing algorithms for representative hardware configurations have been investigated.

1. Text is unlike other data

Most database approaches today have arisen from the database processing needs of business. Even a cursory look at commercial database systems, introductory database textbooks, and traditional database models quickly reveals the database community's orientation towards parts, suppliers and projects.

But not all manipulation of data is best described in terms of operations involving entities and their set-oriented relationships. Consider, for example, a bibliographic entry as follows:


Traditional data modelling is well-suited to capturing the relationships among author, title, and source, but in so doing it completely ignores the essence of the entry itself: its form. As a result, data processing involving the text itself (e.g. extracting a reference such as "[Doe28]" or determining the number of the last page) is outside the scope of most database languages.

We use the term "text-dominated databases" to refer to collections of structured data that are predominantly composed of alphabetical characters. Examples include dictionaries, encyclopedias, almanacs, collections of news clippings, abstracts, legal documents, insurance policies, note cards, and so on, as well as bibliographies. In fact, most of the ideas in this paper arise directly from our involvement in computerizing the Oxford English Dictionary and our previous work on using grammars to define data structures [Gonnet83]. The OED and each of the other text databases rely on highly formatted presentations of textual data that represent carefully composed expressions of facts, not merely an amorphous knowledge base [Tompa86].

In this paper, we present a database model for text-dominated database systems. In Section 2, we introduce "p-strings" which serve as data instances for schemas described in terms of grammars, and we outline the fundamental operators for manipulating parsed text. As is true in relational database environments, the operators can be used to support database views. A few representative examples of the application of p-strings are given in Section 3, and Section 4 describes ensuing research challenges.
2. Schemas, instances, and operators

A text-dominated database is described by a schema expressed as a grammar. Consider, for example, a simplistic bibliographic database with grammar given in Figure 1 (using the syntax of INR [Johnson84], where | separates alternatives, + represents Kleene plus, ? signals that the previous construct is optional, and parentheses indicate grouping). Such a grammar can be used to parse and subsequently represent a collection of data similar to the sample entry given in Section 1; thus the grammar specifies the precise syntax of the entries, which allows it to be used by a parser reading the formatted data [Kazman86].

![Fig. 1: Grammar for a simplistic bibliographic database](image)

A valid database instance, then, contains data that conforms to this schema. Just as numeric data is structured in a business database, string data must be structured in a text-dominated database. Rather than taking the form of tables, hierarchies, or networks, however, grammar-based data takes the form of parsed strings, or "p-strings".

When stored using the schema described in Figure 1, the example from Section 1 takes the form of the parse tree shown in Figure 2, where the leaves are depicted on the left and the root on the right.

Clearly, because the grammar is context-free, the p-string in Figure 2 represents a derivation tree for the sample entry [Ah072]. However, because p-strings represent database instances, they are subject to alteration via operations in a data manipulation language; thus a p-string as an abstract data type is more than just a traditional parse tree.

In what follows, several examples of the operators require sequences of statements; so before continuing with their descriptions, we briefly describe the programming language constructs we shall use.

We have learned from relational vs. other database approaches that users and implementors both benefit from non-navigational languages. Our language, based on the Maple symbolic manipulation language [Char85], has dynamic typing with simple assignment and equality comparison and uses sequential flow, conditional and unconditional statements and expressions, and functions (declared using the keyword proc). We assume that integers and booleans are built-in types (including, for example, provision for integer arithmetic). For notational convenience, \( f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \) applied to a single argument \( x \) represents the call \( f(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n) \) (e.g. \( f(x, y, z) \) denotes \( f(x, y, z) \)).

In the following subsections, let \( E \) be the p-string representing one entry in the bibliography instance as depicted in Figure 2.
2.1. Strings and p-strings

Data conversions between strings and p-strings are fundamental to all other text processing. The operator string returns the complete character string represented by the p-string passed as its argument. Conversely, the operator parsed by takes a string and a non-terminal symbol and creates the instance corresponding to the portion of the schema described by that non-terminal. Thus, for example, 

'Jones' parsed by surname yields

\[
\begin{align*}
J & \rightarrow \text{char} \\
o & \rightarrow \text{char} \\
n & \rightarrow \text{char} \rightarrow \text{surname} \\
e & \rightarrow \text{char} \\
s & \rightarrow \text{char}
\end{align*}
\]

and string ('Jones' parsed by surname) yields 'Jones'. A string is itself a special case of a p-string in which the root is labelled string and the only subtree is the string value. For example,

'Jones' → string

A third operator reparse by takes a p-string and a (partial) grammar and simultaneously for each rule \( L := R_r \cdot \cdot \cdot R_n \) in this grammar replaces each occurrence of sub-p-string \( P \) having root labelled \( L \) by the p-string (string \( P \)) parsed by \( L \). Thus the p-string is re-evaluated according to the new grammar rules.

2.2. Vectors and sets

Many operators described below must deal with collections of p-strings. By integrating the semantics of such collections with the rest of the p-string model, distinct operators need not be developed.

A vector is itself a p-string in which the root is labelled vector. Similarly a set has root labelled set, and any p-string with this label is guaranteed to have no duplicate subtrees. Several functions on vectors and sets would be useful, and these are extended to operate on arbitrary p-strings as well. For example, size returns the number of subtrees (e.g., the cardinality of the vector or set): size ('Jones' parsed by surname) returns the integer 5. To manipulate vectors, the concatenate operator (denoted by a comma) takes two vectors and returns a single vector including all subtrees of the arguments. A third function, mapped onto (cf. SQUARE [Boyce75], Maple [Char85]), takes a function \( f \) with one missing argument together with a p-string \( P \) and returns the p-string that has the identical root label as \( P \) and each subtree replaced by the value of the function when executed with the corresponding subtree of \( P \) passed in place of that argument. For example, if ForceToUpper(\( x \)) is a function defined on p-strings for characters, returning the p-string with uppercase value, then ForceToUpper(\( 'Jones' \) mapped onto ('Jones' parsed by surname)) yields the p-string

\[
\begin{align*}
J & \rightarrow \text{char} \\
o & \rightarrow \text{char} \\
n & \rightarrow \text{char} \rightarrow \text{surname} \\
e & \rightarrow \text{char} \\
s & \rightarrow \text{char}
\end{align*}
\]

2.3. Components

Two operators on p-strings are available to extract the components of an instance. root \( P \) returns the non-terminal symbol at the root of the tree (i.e., root \( E \) returns entry), subtrees \( P \) returns the vector of p-strings that comprise the descendants of root \( P \). A third operator with takes a non-terminal symbol and a vector of p-strings and returns a single p-string that is its composition; thus

\[
\begin{align*}
\text{root} ( n \text{ with } L ) & \equiv n \\
\text{subtrees} ( n \text{ with } L ) & \equiv L
\end{align*}
\]

In fact, because of the interpretation of vectors as p-strings, subtrees merely replaces the root label by the name vector. Conversely, with merely replaces the root label by the non-terminal symbol passed as an argument. This correspondence removes a potentially difficult design problem encountered with representing the *Oxford English Dictionary*: is the dictionary better treated wholly as one parsed string or is it a list of individual parsed entries? Using p-strings, the *OED* can be easily treated in either mode, depending only on the value stored in the root (dictionary vs. vector).

2.4. Selections

The operator in takes a non-terminal symbol and a p-string and returns the p-string having its root labelled by the non-terminal and first encountered when the argument parse tree is traversed by a traditional (pre-order) depth-first search. surname in \( E \) (or equivalently surname in author in \( E \)) thus returns the p-string

\[
\begin{align*}
D & \rightarrow \text{char} \\
o & \rightarrow \text{char} \rightarrow \text{surname} \\
e & \rightarrow \text{char}
\end{align*}
\]

A variant of in is used to retrieve a vector of p-strings representing each subtree subtended by a node labelled by the non-terminal symbol (as encountered in a pre-order traversal). For the example, every name in \( E \) has one subtree; every char in author in \( E \) has seven subtrees; and every initial in \( E \) is the empty vector.

2.5. Transformations

A more powerful operator on p-strings is transduced by, which takes as arguments a p-string and (a part of) a grammar. Simultaneously for each rule \( L := R_1 \cdot \cdot \cdot R_n \) in the grammar, each subtree \( P \) with root \( L \) in the p-string is replaced by \( L \) with \( P_1 \cdot \cdot \cdot P_n \); if \( R_k \) is a terminal symbol, \( P_k \) is the corresponding string; if \( R_k \) is a non-terminal, \( P_k \) is \( R_k \) in \( P \). For any non-terminal untranslated on the left side of a rule, no changes are made. For example, if \( G = \{ \text{author := name \ ' surname; source := journal \ ' year } \} \) then \( E \) transduced by \( G \) yields \( E' \) as depicted in Figure 3.
A second transformation is used to remove unnecessary detail from a p-string. **suppressing** takes a p-string and a non-terminal and returns the p-string with those non-terminals removed — the subtrees for nodes so labelled are connected directly to the parents of the nodes. Thus, given the p-string $E'$ in Figure 3, $(\text{author in } E')$ **suppressing** char yields:

```
J
 o
 h
 a

D
 o
 e

P
 e
 l
 i
 c
 e

9
 2
 8

1

P

entry

author

name

date

digit

surname

roman_text

title

journal

source

year
```

The definition of **suppressing** is extended to suppress multiple nodes; that is, $P$ **suppressing** $(N_1 \ldots , N_n)$ is equivalent to $P$ **suppressing** $N_1 \ldots$ **suppressing** $N_n$.

The third transforming operator, **partitioned by** (cf. SQL's **group by** [Korth86]), provides the facility to perform intra-p-string comparisons. Given a p-string $P$ and a function $F$ applicable to each subtree of $P$, **partitioned by** returns a p-string that groups the subtrees of $P$ by their $F$-values. The resulting p-string is a set of partitions, each of which is represented by a $<F-value, p-string>$ pair (i.e., vector of size two) in which the second component has root label identical to that of $P$ and all subtrees $S$ of $P$ for which $F(S)$ evaluates to the $F$-value in the first component. For example, if $P$ is a p-string representing a complete bibliography and $Date$ is the function

```sql
Date := proc(x)
  date with (every digit in year in x)
  suppressing digit
end;
```

then $P$ **partitioned by** $Date()$ classifies the bibliographic entries by producing a p-string as depicted in Figure 4.

![Fig. 4: Schematic p-string for the partitioned bibliography](image)

The second argument for **partitioned by**, in fact, may be a list of functions, rather than just one. In this case, every sub-p-string is evaluated by each function and grouped by matching values on all functions. As a result, where the $F$-values were stored in the above example, a vector of function values would instead appear in the resulting p-string.

The final transformation operator is **where** (cf. SQL), which takes a p-string and a boolean function and removes complete subtrees for which the function returns false. Thus, $P$ **where** $F$ returns the p-string that $P$ **partitioned by** $F$ would pair with the value true. For example,

```sql
NotSource := proc(x)
  root x <> source
end;
```

$E$ **where** $NotSource()$ produces the p-string depicted in Figure 5.

3. Illustration from the *Oxford English Dictionary*

To demonstrate some of the facility of p-string databases, consider this simplification of the *OED*:
where the following non-terminals are used:

- **hwp** - headword group, used to identify an entry
- **etym** - etymology, indicating the word's evolution
- **sen** - sense, a meaning of the word
- **quote** - quotation, an illustrative example taken from a printed source, including the date of the quotation, the source, and the text itself
- **senbank** - sense bank, a sense together with its illustrative quotations

Such a p-string can result from the application of the following grammar:

\[
\begin{align*}
dictionary & : = \text{entry}^+ ; \\
\text{entry} & : = \text{hwpg etym? senbank* sen*} ; \\
\text{senbank} & : = \text{sen+ quote+} ; \\
\text{quote} & : = \text{qdate qsource qtext} ; \\
\end{align*}
\]

One application for the OED is to examine all Anglo-Norman influences. Which words derive from Anglo-Norman terms is determined primarily from the first language (other than Old or Middle English) appearing in the etymology. For example, the etymology

\[\text{a. F. aromatique (14th c.), ad. L. aromatic-us, a. Gr. ...}\]

indicates a word that was adopted from French, which had borrowed it as an adaptation of a word from Latin, which, in turn, had adopted its word from Greek. Therefore a finer grammar for etymology, identifying names of languages as well as delimiters and other text, must be used.

EtymG :=
\[
\begin{align*}
\text{etym} & : = \text{lang? (text delim lang)* text} ; \\
\text{lang} & : = (\text{OF.}, \text{AF.}, \text{F.}, \text{L.}, ...) ; \\
\text{text} & : = (\text{char | italic-char}+) + - \text{lang} ; \\
\text{delim} & : = (\text{. . . . . . . . .}) ; \\
\end{align*}
\]

where ‘OF.’, ‘AF.’, ‘F.’, and ‘L.’ are the OED’s abbreviated forms for Old French, Anglo French, French, and Latin, respectively.

In fact, such a grammar will have been applied by the publisher in preparing the dictionary, but, again for illustration, we will assume that the stored form of the etymology does not identify languages. (It is certain that there will be aspects of the dictionary that are not identified adequately for specific applications.) Thus, in this example, the etymologies must be reparsed by the user:

\[
\begin{align*}
\text{firstlang} & : = \text{proc(x)} \\
E & : = \text{etym in x} ; \\
L & : = \text{lang in (E reparsed by EtymG)} ; \\
\text{string}(L) & : = \text{end} ; \\
\end{align*}
\]

For any p-string P, firstlang(P) finds the first occurrence of etym contained in P, reparses according to the grammar, extracts the first lang in the result, and returns the string subtended by that p-string. For the etymology

\[\text{a. F. aromatique (14th c.), ad. L. aromatic-us, a. Gr. ...}\]

re parsing produces a p-string associating non-terminals with strings as follows:

<table>
<thead>
<tr>
<th>non-terminal</th>
<th>associated string</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>‘a.’</td>
</tr>
<tr>
<td>delim</td>
<td>.</td>
</tr>
<tr>
<td>lang</td>
<td>‘F.’</td>
</tr>
<tr>
<td>text</td>
<td>‘aromatique (14th c.), ad.’</td>
</tr>
<tr>
<td>delim</td>
<td>.</td>
</tr>
<tr>
<td>lang</td>
<td>‘L.’</td>
</tr>
<tr>
<td>text</td>
<td>‘aromatic-us, a.’</td>
</tr>
<tr>
<td>delim</td>
<td>.</td>
</tr>
<tr>
<td>lang</td>
<td>‘Gr.’</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>

and firstlang therefore returns the string ‘F.’

Thus we can identify those entries having etymologies beginning with ‘AF.’ or ‘OF.’ or ‘F.’ However, a word marked as French is Anglo-Norman only if it entered English before 1500: for entries with first language ‘F.’, the earliest quotation date must be examined as well.

Assume that there is a function year which returns an integer representing the expected value for the year for any (formulaic) date appearing in the OED structure qdate, the date associated with a cited quotation [Gonet 86]. Furthermore, assuming that the function min returns the smallest integer occurring in a vector of integers, the following procedure finds the earliest quotation date in a p-string:

Proceedings of the 13th VLDB Conference, Brighton 1987
earliestquot :=
proc (x)
  q := every qdate in x ;
  if size q > 0
    then min (year() mapped onto q)
    else 0
  fi
end ;

Finally, we can identify those entries of the dictionary representing Anglo-Norman influences:

IsAN := proc (x)
  firstlang = 'AF.'
  or firstlang(x) = 'OF.'
  or firstlang(x) = 'F.'
  and earliestquot(x) < 1500
  and earliestquot(x) > 0
end;

AngloNorman := NewOED where IsAN();

Because the complete entry is not always desired, interesting structural components may then be selected. For example, a researcher conducting exploratory analysis of Anglo-Norman influences, might wish to suppress (temporarily) the meanings of the English words that are derived from Anglo-Norman terms. A special-purpose grammar can be defined to identify precisely those components of the p-string that are of further interest:

ExtractG :=
  { entry := hwgp etym? senbank* ;
    senbank := quote+ ;
    quote := qdate ;
  }

ExtractG can be used to preserve only that part of the structure containing hwgp, etym, and qdate, after which the non-terminal symbols senbank and quot may be considered to be superfluous, in this case they can be suppressed in the p-string:

(AngloNorman transduced by ExtractG)
suppressing {senbank, quot}

As a follow-up, it might be useful to classify the Anglo-Norman entries of the OED by the source language and decade of first use in English. To this end, we use the expression

AngloNorman partitioned by
  (firstlang(), decade())

where

decade := proc(x)
  10 * floor(earliestquot(x)/10)
end;

to obtain the structure in Figure 6.

Fig. 6: P-string for the partitioned AngloNorman dictionary

4. Further research challenges

In developing the p-string model for text-dominated databases, we have attempted to provide for text systems what the relational model provides for commercial systems. The operators that we described have been found to be useful in formulating solutions to the majority of queries that have been posed so far for the Oxford English Dictionary research being undertaken by several academic scholars as well as by dictionary editors. In addition, we have formulated partial solutions to diverse problems involving other text domains including transforming two incompatible bibliographic formats to a common one in order to merge their entries, converting computer permission files into electronic mail directories, producing mailing labels from online name and address files, and so on.

A phenomenon that rarely appears in computer science results from our work on text-dominated databases. Unlike much other work, research involving p-strings has the luxury of having many interesting problems already identified as a result of accomplishments in the areas of programming languages, language theory, and conventional database management. For example, it is clear that questions of support for security, concurrency, views, and data distribution arise in text-dominated databases — first approximations to solutions can be derived from research in other areas, but almost certainly the differences of text from commercial data will require modifications to existing solutions; in some cases the required modifications are expected to be significant. Text-dominated databases should also be supported by semantic models, and, at first, the functional data models seem appropriate. However, our preliminary efforts at modelling text databases have repeatedly floundered because of the amorphous nature of textual
"entities." Similarly, a theory of p-string database systems will likely arise from database theory, logic, and automata theory. For example, is there a useful parallel to normal forms? How should constraints on a p-string instance be expressed and manipulated?

Probably one of the richest open areas is how to implement the data type p-string efficiently. Several potential representations come to mind immediately: explicit representation of the parse tree as a recursive data structure encoded by pointers to siblings and subtrees — with or without parent pointers; implicit representation by SGML-like start and end tags embedded in the text string itself [ISO85, Gonnet86]; and tables mapping non-terminal symbol names to records representing corresponding subtrees in the p-string. Although some operators are superficially similar to traditional database operators (cf. every.in and where vs. 'select' or 'project' and partitioned by vs. 'group by'), as a challenge, we claim that an encoding of p-strings using any of the conventional database models (including implementational as well as semantic models) would be inappropriate: the amount of application code to be written to support query and update would be comparable to the amount of code required using an implementation language such as C.

One interesting question concerns the power of the language described, particularly its power relative to relational database query languages. Using a representation of relations defined by the following grammar

\[
\text{relation} := \text{tuple} * \; \text{;}
\]

\[
\text{tuple} := \text{attribute} + \; \text{;}
\]

\[
\text{attribute} := \text{name} \; \text{value} \; \text{;}
\]

José Blakeley has implemented select, project, Cartesian product, union, difference, and attribute renaming. It is perhaps instructive to view the simulation of Cartesian product, assuming for simplicity that the attribute names are all distinct.

\[
\text{TupleXTuple} :=
\]

\[
\text{proc} (t_1, t_2)
\]

\[
\text{tuple with (subtrees } t_1, \text{ subtrees } t_2) \;
\]

\[
\text{end};
\]

\[
\text{TupleXRelation} :=
\]

\[
\text{proc} (t, R)
\]

\[
\text{TupleXTuple}(t) \text{ mapped onto } R \;
\]

\[
\text{end};
\]

\[
\text{CartesianProduct} :=
\]

\[
\text{proc} (R_1, R_2)
\]

\[
\text{TupleXRelation}(R_1, R_2) \text{ mapped onto } R_1 \;
\]

\[
\text{end};
\]

In a similar manner each operation of relational algebra can be implemented, thus showing that the p-strings operators form a relationally complete set. Is there an alternative, preferable encoding of a relational database as a p-string? Is there a suitable non-procedural language that is able to express the same computational power as our p-string algebra?

An interesting open question in this respect is the effect of the type of the defining grammars used. In all the examples we have encountered, we have used a BNF-like notation for describing languages which are, in fact, regular. What is the power of the p-string language when the grammars define non-regular context-free languages? What changes must be incorporated to accommodate context-sensitive or Type 0 languages? In what application areas would these be useful?

Acknowledgements

We wish to acknowledge the contributions of many of our colleagues with whom we discussed these ideas, particularly deserving of mention are José Blakeley, Howard Johnson, Ian Munro, and Sylvia Osborn. The search for Anglo-Norman terms in the OED is part of a research project being carried out by Delbert Russell. We also acknowledge the financial support of the University of Waterloo and the Natural Sciences and Engineering Research Council of Canada via the University-Industry Program Grant CRD-882.

References


