A SYNTAX DIRECTED DATABASE SYSTEM

by

Dennis Heimbigner*

CU-CS-289-85 January, 1985

*Department of Computer Science, University of Colorado, Boulder, Colorado 80309

A Syntax Directed Database System

Dennis Heimbigner
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

ABSTRACT

The syntax-directed database system provides a relational interface to text files. It allows text files to maintain their textual format while still allowing sophisticated access via relational query languages. The SDDB operates by parsing the text file using an attributed grammar. The attributes of the grammar couple the text file to the attributes of a relational schema. Relations are stored as tuples containing pointers to the corresponding strings in the text file.

Introduction

This paper describes a database system that can directly manipulate databases maintained as text files. This system is termed the Syntax-Directed Database System (SDDB). The SDDB supplies a relational database interface to data while still keeping it in text form. In this way, the data can continue to be manipulated by normal text-processing software. The SDDB could usefully manipulate a number of existing classes of text databases:

(1) The messages produced by electronic mail systems,
(2) The VLSI designs of CIF format files,
(3) Program files of subroutines and functions,
(4) The password and termcap files of Unix systems,
(5) Scribe [Reid 80] bibliography files.

The Syntax-directed Database Architecture

Externally, the syntax-directed database system appears to a user much like a normal relational database system. Internally, the SDDB has two basic components: a database and a parser. There is a (modified) relational database that serves as the major interface to the user. The database system, in turn, uses a syntax-directed parser to determine the structure of a text file. When a user wants to manipulate a text database (or databases), he specifies the file containing the data and the structure

of that file. The structure is specified by a grammar and a relational schema. The file is parsed and converted to a format suitable for manipulation by the relational database. The user employs the operations of the database model to retrieve information from the text, and to modify the data in that same file. When he is finished, the user releases the text file and uses it as input to other programs, just like any other text file.

A Mail Example

The example to be used in this paper concerns a file of mail messages, such as is generated by the mail program under Unix. Figure 1 shows a sample mail file consisting of two mail messages. Each message has a destination list (tagged by the keyword "TO"), a sender, a date, a subject line, and a body. The body is a series of zero or more lines of text. The end of the body is indicated by a line containing only a single dot. Obviously this is a greatly simplified form of message, but it will suffice to demonstrate the features of this system.

Using the syntax directed database on this mail file would allow a user to ask questions like: "Find all messages sent to me by Paul on December 21, 1984." The

---

```
TO: Dennis, Roger
FROM: Paul
DATE: December 1, 1984
SUBJECT: Lunch
How about lunch on Friday?
.
TO: Paul
FROM: Dennis
DATE: December 2, 1984
SUBJECT: Re: Lunch
Sounds good.  How about at 11:30?
.
```

Figure 1. Example Data File

---

answer could be retrieved by the following relational algebra query:

$$\sigma_{sender = Paul \wedge date = December\ 21,\ 1984}(Mail))$$

Schema and Grammar

The data format for a particular text file is specified by a relational database schema and a grammar. The schema specifies the format of the relations that correspond to the data in the text file. A schema consists of a collection of relation names. Each relation name is followed by a parenthesized list of attribute names. For this version, all attributes are of type string.

The relation schema to be used for the mail file is

MAIL(destination,sender,date,subject,body).

The single relation is named MAIL, and it has five attributes: destination, sender, date, subject, and body.

The grammar defines the actual format of the text file as well as the procedure for translating from the text to the relations. The Unix YACC parser generator system is the basis for the SDDB parser and associated grammars. Each grammar is in the standard YACC format but with some standardized semantics for converting parse trees to relations.

At the moment, the grammar is expected to use a standard set of lexemes. Obviously this not the most general case. As soon as practical, the LEX lexical analysis system (or something equivalent) will be added to the system. Thus the format of the text file will be specified by a combination of a grammar and regular expressions.

Figure 2 shows the grammar for our mail example. For the moment, ignore the actions in curly brackets, and concentrate on the grammar. It indicates that a mail file (mbox is the start symbol), consists of a list of zero or more messages. Each message consists of a tolist, a sender, a date, a subject, and a body. Suitable tags are

placed in the grammar to disambiguate the text. The tolist, in turn is a production to match a comma separated list of destinations. A subject is a everything from the tag "SUBJECT:" up to the end of the line. The date in this grammar is simplified to just be a line of text. The body is a list of lines, the last of which must contain a single dot character (production "endbody").

The relational schema is connected to the grammar via the attributes which are associated with certain non-terminals in the grammar. These attributes correspond to the relation attributes in the schema for the file. Each attribute has an associated ordered list (value list) that is separate from the database, and it is used for the temporary storage of parsed values. Whenever a non-terminal is successfully used in the parse, the string matched by that non-terminal is appended to the list for the attribute associated with that non-terminal.

In our example (Figure 2 again), we can see that the semantics portion (in curly brackets) of some of the productions contain references to the function "assign-attribute". This function takes as its argument the name (as a string) of an attribute in the relations schema. For example, whenever the production "destination" is reduced, it will invoke "assign-attribute" to collect the string matched by the production. The function will place that string in the list associated with the attribute named as its argument ("destination" in this case). Thus, as an instance of "tolist" is parsed, it will collect a sequence of destination names: one for every reduction of the destination production. Similar actions occur when the productions "sender", "date", "subject" and "body" are used (see figure 4a).

Productions may have two additional kinds of tags associated with them: clear lists and generators. The clear list is a list of attributes. It specifies that the value lists associated with the specified non-terminals should be cleared. A generator (see below) is a function that is invoked whenever the associated non-terminal is reduced

```
mbox            : msglist ;

msglist         : /*empty*/
                | msglist msg ;

msg             : TO ":" tolist NEWLINE
                  FROM ":" sender NEWLINE
                  DATE ":" date NEWLINE
                  SUBJECT ":" subject NEWLINE
                  body ;
                        { generate("genmsg");
                          clear("destination")
                          clear("sender");
                          clear("date");
                          clear("subject");
                          clear("body");
                        } ;

tolist          : sender
                | tolist "," destination ;

destination     : NAME
                        {assign-attribute("destination");} ;


sender          : NAME
                        {assign-attribute("sender");} ;

date            : LINE
                        {assign-attribute("date");} ;

subject         : LINE
                        {assign-attribute("subject");} ;

body            : bodylines endbody
                        {assign-attribute("body");} ;

bodylines       : /*empty*/
                | LINE NEWLINE bodylines ;

endbody         : "." NEWLINE ;
```

Figure 2. Example YACC Grammar

during the parse. These reductions mark the places where complete tuples can be constructed from some set of value lists. Thus, as parsing proceeds, values are temporarily collected in the value lists, and at certain points, the collected values are converted to tuples, inserted into the database and the lists selectively cleared.

Figure 2 shows that the attribute lists for destination, sender, date, subject, and body are all cleared as part of the reduction of the production "msg". The clear cannot occur lower in the tree because all three attributes are needed to generate tuples in MAIL. It cannot occur higher because these attributes change for every new message in the message file.

Generators

A generator is a simple function over a set of attributes. It is invoked to convert attribute lists into a set of complete tuples for some relation. It is constructed from some primitive functions that perform the most common kinds of constructions. At the moment, there are two such primitive functions: cross product and dot product. The cross product is used to combine two attributes that have a many to one relationship. Usually, one of the attributes has only a single value that is to appear with all of the values in the second set. In the text file, this often corresponds to a header that appears once followed by a list of items. Dot product is used to match up values in two lists of equal lengths. The equal length criteria is usually guaranteed by the structure of the grammar. The dot product is used for those situations where two or more columns of related items appear.

Figure 3 shows a generator for converting the attributes destination, sender, date, subject, and body into tuples for the MAIL relation. The generator, named genmsg, takes the dot product of sender, date, subject, and body, and then crosses that with the destination attribute. This means that a message with N destinations will produce N tuples; each tuple will have one of the destinations combined with the common

sender, date, subject, and body values.

Implementation Structure

There are basically two ways to store the relations produced as a result of the parse. First, one might build tuples containing the actual data and store those tuples in the database. This introduces a consistency problem between the relation and the text file. Changes to the relations are hard to propagate back to the text file. It also violates the idea that the text file is the primary repository of the data.

The second storage possibility, and the one used here, involves storing pointers into the text. Thus, the tuples do not actually contain the data. Rather, each field in the tuple contains the length and position of the field value in the original text file. The basic database functions and structures can operate fairly normally, including indices (b-trees are provided). The only difference is the introduction of one level of indirection for accessing actual values from the text file.

This second implementation has the disadvantage of speed, but has two advantages: it takes less space and it maintains consistency between the database and the text file.

A Processing Example

When the Mail relation schema and the grammar of figure 2 are applied to the data of figure 1, the result will be three tuples: two from the first message (it has two destinations) and one from the second. Figure 4a shows the value list associated with

---

define genmsg(MAIL) = destination X (from • date • subject • body)

Figure 3. Example Generator

---

each attribute after the parse of the first message, but before the generator has been invoked.

Figure 4b shows a user's view of the contents of the relation MAIL after the appropriate generator has been invoked on the attribute lists of figure 4a. Figure 4c diagrams the actual implementation in terms of pointers into the text.

Once the tuples from the first message are generated, the clear-lists are invoked (as a side effect of the parse) to reset the attribute value lists. Then the second message is parsed, one more tuple generated, and again the clear-list applied. The final result is three tuples in the relation.

Update

The proper definition of update for an SDDB is still somewhat of an open question. When a field in the database is changed, the corresponding value in the text file must ultimately be changed. There are three problems associated with this. First, if several tuples point to the same text, but only one tuple is altered, what, if any, should be the affect on the other tuples? Second, if an indexed field of a tuple is modified, how should the indices be updated? Third, how should modifications be handled that are larger or smaller than the original text?

We have chosen to make changes to one tuple affect all tuples covering overlapping text. It appears that the side-effects are often desirable because it mimics the effect of editing the original text file. In figure 4c, for example, editing the body of the message in either tuple 1 or tuple 2 will affect the other. But this seems to be a reasonable action to take.

The second question (index updates) is a problem because modifying one field in one tuple may affect a number of other tuples. If that field is indexed, then the tuples will be in an incorrect location in the index. The b-trees that we are using store record

Attribute Lists

destination	sender	date	subject	body
"Dennis" "Roger"	"Paul"	"December 1, 1984"	"Lunch"	"How about lunch..."

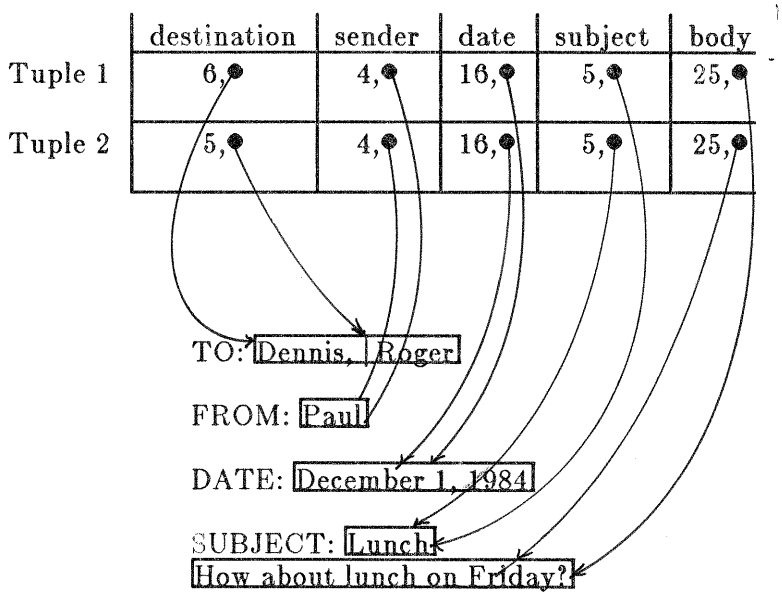
Figure 4a. Processing Message 1

Relation MAIL

destination	sender	date	subject	body
"Dennis"	"Paul"	"December 1, 1984"	"Lunch"	"How about lunch..."
"Roger"	"Paul"	"December 1, 1984"	"Lunch"	"How about lunch..."

Figure 4b. Tuple Generation for MAIL

Relation MAIL



Mail Messages Text File

Figure 4c. Implementation Structure

pointers with the index value. It is easy to find all the tuples potentially affected by a change by indexing into the tree using the old value. Then all of the records associated with that value can be removed from the tree and re-inserted.

The third question (the size change for fields) is handled at the moment by a separate changes file to hold the modifications to the text file. When a field value is changed, the new value is inserted somewhere into the changes file. Each change is associated with the place in the text that it modifies. When the text file is released from the SDDB, these modifications are merged with the original to produce a new version. The alternative would have been to modify the original on the fly, but it was felt that this would involve too much text transfer to be practical.

Using a changes file presents an additional problem: making sure that overlapping modifications all refer to the same change. Potentially, this entails a cross check between the text file and the changes file at every reference to a field. To avoid such frequent checks, the text file field value is modified to contain a special character to signal a change. Whenever a tuple field value is read from the text file, it is checked for this special character. If it is found, then the changes file is referenced to obtain the modified value. This places two limits on the form of text files: all fields must be at least one character in length, and there must be some character not used in the text file. Neither restriction appears to be a serious problem for any reasonable text file.

Modifying the original text is undesirable if, for example, that file is to be concurrently read by some other user. This can be handled by copying, during parsing, the text file to a local copy, which is the one that is actually modified. When the modified version is released, it replaces the original under simple file locking.

When the text file is released from the SDDB, any modifications must be merged with the original file. By keeping the changes file indexed by location (using some form of b-tree again) the merge can be performed in one pass over both files. This

involves scanning the leaves of the tree in sequential order and at the same time scanning the text file. For every change in the tree, the text file scan is advanced to the point of change, and then the modification is made.

Efficiency

It is clear that an SDDB can never be as efficient as a normal database system. It is trading speed for the ability to maintain the original text file. Nevertheless, it is important to make the system as efficient as possible. Re-use of the parse information is an important efficiency feature.

In the simplest case, if the text file is parsed once by the SDDB then released, and then used again by the SDDB, no re-parse should be necessary (the old parse can be re-used). To catch such situations, the database produced by the first parse is kept around (cached) until it is invalidated.

A cache can be invalidated whenever the text file is modified. If the modification is external to the SDDB, then there is nothing that can be done; the old database is purged, and the text file is re-parsed. This case can be detected by recording the modification time and date for a file.

The text file is also modified by the database itself when it is merged with the changes file (see discussion of update). These modifications can be propagated back to the database structures (tuples and indices) with some effort. Each change (unless the size is constant) alters the location of all fields after the one that is modified. Thus many, perhaps all, of the tuples in the database need to be modified. Rather than perform all of those changes, the original text file and the modifications file are kept intact. If later the modified file is again given to the SDDB, and it has not been modified externally, all of the old information can be kept and used. There may be a series of new versions, all produced from the same original by an evolving changes file. At some point, the cost of using a large changes file will outweigh the cost of re-

parsing, and the changes file can be purged.

Errors

Errors in the text are a serious problem for syntax directed parsers. If the grammar is recursive, then a missing marker may not be detected until the end of the text file is encountered. Fortunately, it appears that most SDDB grammars do not require arbitrary nesting, and so it is usually possible to detect errors fairly quickly through the occurrence of specific strings marking the beginning of top-level structures. Much of the work on compiler error recovery is applicable here, and some simple errors can be automatically detected and corrected. In other cases, there is little that can be done except to isolate the offending text and continue parsing the rest. As a last ditch effort, the SDDB can allow the user to make a quick change to the text in the middle of parsing.

Related Work

There is a large body of work on syntax-directed compilation of programming languages (see [Aho 73], for example). But compilers typically do not allow a user to dynamically manipulate and modify the program file. The SDDB, however, is explicitly designed to handle evolving and modifiable files.

Text databases are usually handled by so-called Document retrieval systems [Salton 83]. They emphasize whole text retrieval based on keywords; no sophisticated structure (relational or otherwise) is imposed on the text.

Some work has been done on using database systems for word processing [Stonebraker 83] by forcing the text to be stored in the database. In some sense this is the opposite of an SDDB which extracts structure from the text rather than tearing apart the text and placing it into a traditional database.

A number of string manipulation languages [Aho 79, Griswold 79] have been designed for manipulating text files. They include varying levels of pattern matching, and in the case of AWK, there is even a rudimentary record and field facility. The problem with these languages is that they are too low level. They do not provide the powerful interfaces that characterize relational databases (such as the relational algebra), but which are relatively easy to provide in the SDDB.

The syntax-directed database is also being used as part of another project [Heimbigner 84a]. This use involves extracting data from the text output of other database systems rather than from files. When the SDDB is combined with some other features the result is a system that can extract and combine information from external databases without requiring any modifications to those database systems.

Future Work

As a first step, a prototype of the SDDB is being constructed by modifying an existing relational database system. As mentioned before, YACC serves as the basis for the parser. Obviously, YACC is not "user-friendly", and, eventually, a better interface (or possibly even a new parser) must be provided.

The concept of a syntax-directed database points to some interesting avenues for database research. For example, there is no reason to restrict the user view to a relational model. The syntactic model described in [Heimbigner 84b] is a variation that allows one to store complete parse trees. It also provides a manipulation language for traversing these trees in useful ways.

References

[Aho 73] Aho, A. V., and Ullman, J. D., *The Theory of Parsing, Translation, and Compiling,* Volume 2, Prentice-Hall.

[Aho 79] Aho, A. V., Kernighan, B. W., and Weinberger, P. J., "Awk - A Pattern Scanning and Processing Language," *Software - Practice and Experience,* 9(4):267-280, April 1979.

[Griswold 79] Griswold, R. E., Hanson, D. R., and Korb, J. T., "The Icon Programming Language: An Overview," *SIGPLAN Notices,* 14(4):18-31, April 1979.

[Heimbigner 84a] "Towards an Integrated Environment for Accessing External Databases", *Proceedings of the Second ACM-SIGOA Conference on Office Information Systems,* Toronto, Canada, 25-27 June 1984.

[Heimbigner 84b] "A Syntactic Database Model", University of Colorado, Boulder, Department of Computer Science Technical Report CU-CS-283-84, December 1984.

[Reid 80] Reid, Brian K., *Scribe: A Document Specification language and its Compiler,* Ph. D. Thesis, Carnegie-Mellon University, also available as Technical Report CMU-CS-81-100, Department of Computer Science, Carnegie-Mellon University, October 1980.

[Salton 83] Salton, G. and McGill, M. J., *Introduction to Modern Information Retrieval,* McGraw-Hill pub., 1983.

[Stonebraker 83] Stonebraker, M., Stettner, H., Lynn, N., Kalash, J., and Guttman, A., "Document Processing in a Relational Database System", *ACM Transactions on Office Information Systems,* 1(2):143-158, April 1983.