AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

Data Extraction Tools

By D. G. BELANGER and C. M. R. KINTALA*

(Manuscript received April 9, 1984)

Data analysis includes the acquisition of data from a wide variety of sources, using different media and ranging in size from a few bytes to hundreds of millions of bytes. In the case of large data sets, the problem reduces to finding an efficient way for an analyst or other user to extract useful subsets from the source data with minimal programming knowledge and effort. Since different sources of data often have entirely different protocols, interfaces and procedures for access, the problem is also to reduce the complexity of data access by hiding this variety from the user. We describe a table-driven program generation system that provides a uniform interface to the analysts for requesting portions of data from any source. The system then generates a program that executes on the source system and extracts the requested data. The table-driven nature of the generator can be used to modify the style of the programs being generated. The tables are, in fact, target program abstractions specified in a high-level language. We now have tables that encode efficient C programs to extract data from IBM standard label tapes on *UNIX*™ systems and COBOL/DL-I/JCL programs to extract data from Information Management System databases.

I. OVERVIEW OF THE PROBLEM AND SOLUTION

The acquisition of data is a crucial part of the data analysis process. This data may be small enough in volume that it can be entered by hand by a single user (e.g., parameters to a program or a small data file); it may be data (e.g., survey data) entered manually by a variety of people; or it may be large volumes of data contained in the opera-

* Authors are employees of AT&T Bell Laboratories.

tional databases of the company (e.g., data on toll calls and charges from a specified area). A set of tools is described in this paper that addresses the problem of extracting, in a useful manner, data from high-volume sources (within or outside of the company). Specifically, there are tools for extraction of data from Information Management System (IMS) databases [Information Management System to *UNIX* System (IMX)] and from tapes generated on IBM computers (TTU). These tools use a single syntax and are resident on a *UNIX* system. The tools are built with a flexible code generation technique which allows for easy addition of new data sources. The key ideas in this process are the following:

1. The analyst typically does not have control over the source environment of the data. In particular, the computer system, database management system and mode of access (or distribution) of the data are likely specified by the administrator of the data.
2. The user should see a uniform and high-level language for data extraction. The details of various database systems and operating systems should be hidden from the user, fostering the feeling of working entirely within a single analytical environment.
3. The forms in which data are available are varied and changing. Any system of this sort must allow for new database management systems or other sources to be added easily.
4. The use of the system should be simple enough that the user is encouraged to extract only the amount of data that appears to be needed, knowing that more or different data can be easily obtained.
5. The volume of data available, and necessary, may be very large ($>10^8$ bytes per data set). Thus efficiency of extraction is critical.

Our approach to this problem was to design (1) a program generation system with a very high-level interface so that an analyst can easily phrase the required data request (i.e., no programmer help needed), and (2) a table-driven code generation system so that the system could be set up to generate code for several different modes of data extraction. In fact, we now have two tables for data extraction. One is for arbitrary IMS databases, for which the system writes programs using JCL, COBOL and DL/I languages; and one for extraction from Extended Binary Coded Decimal Interchange Code (EBCDIC) coded tapes (and translation to ASCII or binary code), for which C programs are written. In each of these cases the program generated is specifically written for the request made. The program is very fast, especially in the tape case (in fact considerably faster than one would expect from a typical programmer asked to do the same job).

Because of its usefulness to analysts, the program generation system and the two tables (for IMS and tape extraction programs) are often used in conjunction with other analytical tools, some of which are described in this issue.

II. CURRENT STATUS

Versions of the program generation system for translation from IMS databases and from EBCDIC tapes are running on the AT&T *UNIX* Operating System V. The current versions are available to users within AT&T. These generators are currently being used routinely on data sets on the order of 150 megabytes. Much larger data sets are also being processed using these tools.

III. HISTORY AND MOTIVATION

3.1 The problem

During the process of data analysis, a wide variety of source data must be obtained and analyzed. This data may come from many different sources on different media (e.g., paper, magnetic tapes) and may range in size from a few bytes to hundreds of millions of bytes. The data itself may originate and be stored in a wide variety of computers using an even wider variety of database management (or file) systems. Often, these systems are part of a corporation's operations systems (e.g., accounting, sales operations) and are not accessible on a time-shared basis. When they are accessible, the mode of access is typically dictated by the database staff rather than by the analyst. The problem of data acquisition in these cases reduces to that of efficiently accessing a very large data set of a known type (e.g., an IMS database or a foreign tape) and extracting a defined subset of that data. In order to access such data sets, the analyst requires that a single, preferably high-level, language be available for making these requests.

The problem we are addressing then is how an analyst can routinely obtain data that is already in machine-readable form without learning new languages for the data retrieval and operating systems of foreign computers, and with the flexibility and efficiency needed to explore the data.

3.2 Existing tools

There are, of course, many tools which address parts of the data extraction problem. In this section we discuss the relationships between the IMS-to-*UNIX*/Tape-to-*UNIX* (IMX/TTU) system and other tools available to users of the *UNIX* operating system to manage data acquisition of large amounts of data.

If we consider the tools available on the *UNIX* operating system for the translation of foreign tapes, we see that a combination of `dd` and awk, or in some cases dd and grep, can be used.[1] This approach will handle simple, character-oriented (i.e., not packed or binary) data. In order to do field selection, the user must write a script in awk. To do record selection on patterns, grep may be used. This method is not fast enough for very large data sets, even with a much faster than standard version of dd in use in our organization. In general TTU provides more functionality and is faster.* In addition, it takes advantage of efficiencies that can be gained by reference to the user's actual needs (e.g., translate only the requested fields) and by allowing users to request calls to their own processing routines. Finally, this method anticipates future expansion of needs and hiding their solutions within the current extraction language.

In the case of interfaces to database management systems (in this case IMS), the problem is not the absence of some good extraction systems (e.g., RAMIS II†). Instead, it is the lack of control over the environment in which the target data resides and the lack of uniformity in the extraction system's languages. Our approach has been to generate extraction programs in COBOL, initially, to provide a reasonably global coverage of installed IMS databases with IMX. This allows access to databases without access to a specific extractor. On the other hand, we have left open the option of adding program generation tables to use higher-level interfaces to IMS (or other database management systems). For example, provision of tables for RAMIS II and for ADABAS‡ is being considered. The point, then, is not that we need to raise the level of these languages but that the analyst needs a single interface (i.e., does not want to spend time relearning several different interfaces) and that our approach will provide that single interface for systems with low-level-interface languages (e.g., tapes translated in the C language) as well as those with higher-level languages.

IV. DESIGN APPROACH

In our system of data extraction tools, we take a table-driven program generation approach to retrieve data from large source data-

* For example, choosing 68 percent of 2 million 64-byte records using dd | egrep ran at about 136 seconds per megabyte. In addition, downstream processing of the data must read all 64 bytes per selected record. Using TTU, extraction times were about 8 seconds per megabyte when reading the entire record. In this example, only 24 of the 64 characters in each record were required. This reduced TTU time further to 4.5 seconds per megabyte. Of course, there was no change in dd | egrep. In this example, total system and user time decreased from 6.1 hours for dd | egrep | C program to 0.1 hour for TTU with processing option. Even when using the entire record, the difference is 8.4 hours to 0.2 hour. In general, improvement factors of 10 to 100 have been observed, depending on the data requested.[2]

† Trademark of Mathematica Products Group Inc.

‡ Trademark of Software AG of North America.

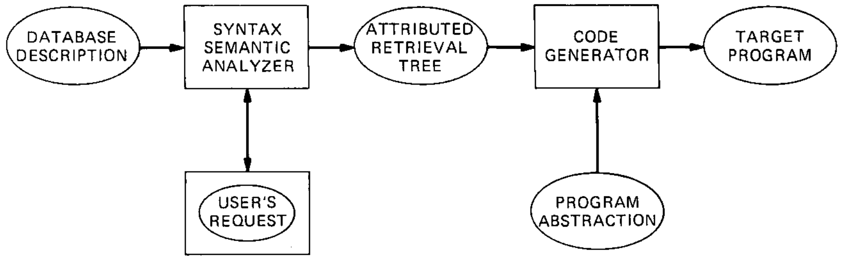

Fig. 1—System architecture.

bases. The architecture of this approach is depicted in Fig. 1. A dictionary file describing the source database is first created. This file provides the information for the system's display and part of the code-generator's decision process. The analyst requests the portion of the data to be extracted from this source in a high-level extraction language. The program generator analyzes this request for syntactic and semantic correctness and produces an "attributed retrieval tree" of this request. It then traverses this tree to translate the user's request into an equivalent program for the target database system using a table containing an abstraction of the extraction programs. This abstraction of the data extraction programs for the target system is written only once for each target database system in the language KS, a program abstraction language for the program generator.

This method of translating users' requests for data in a high-level language into complete programs for execution on the target database systems is similar to, albeit simpler than, the process of compiling. The attributed retrieval tree produced by the syntax and the semantic analyzer is similar to the parse tree produced by the front end of a compiler. Interpreting and making structural expansions of the constructs in the target program abstraction with the retrieval tree as the basis is analogous to a combination of the interpretative and the table-driven methods used in retargetable code generators for compilers.[3] A more formal treatment of the analogy between query language translators and compilers can be found in Ref. 4. By applying compiler technology to the database problems we hope to provide new insights into these problems and obtain new tools.

Aside from being similar to compilation, this table-driven program generation approach has several advantages in practice. Data extraction programs tend to be routine but tedious in character. Often, an application programmer working on a database system acquires an "existing" or a "sample" extraction program and changes it with the appropriate information about the new extraction request at the "appropriate" places in the original program to produce a new program for the new request. This "knowledge," about how to write the new program, is encoded into the target program abstraction written in the KS language. The program generator can then "automatically" produce the target program given sufficient information about the request and the source database. Thus, this approach (1) provides high-level interfaces to the analysts for extracting data from databases, and (2) automates the process of programming to extract data.

By making the program generation table-driven, source database descriptions and target program structures can be changed easily. For example, in TTU, the tool that generates programs to read tapes, we have been able to add more functionality and speed to the system by simply incorporating some changes to the KS table without any recompilation of the tool itself. This approach separates the basic code generation algorithm from the details specific to the underlying database structure or the target program structure. This also allows us to easily change the tape record structures or the programming language, say from C to assembly.

V. SYSTEM DESCRIPTION

As explained in the previous section, we have a table-driven program generation system that can be easily targeted to generate programs, in a variety of programming languages, to extract data from a variety of source databases, such as tapes or IMS database systems. TTU, the tool we assembled from this system for generating C language programs to extract and translate data stored in EBCDIC format on tapes, will be described in this section to illustrate our system.

5.1 User interface

Any user wishing to extract data from tapes using TTU goes through the steps illustrated in Fig. 2. For any tape type,* the user or a data administrator must create a dictionary (i.e., a file describing the tape). A syntax-directed editor based on the "age" system[5] is provided for this purpose. It guarantees the syntactic correctness of the dictionary. Alternatively, any text editor may be used to create the dictionary by following the defined structure. Once the dictionary is created, users can specify the record conditions and fields to be selected. TTU uses this information to generate a program. The generated program performs the selection, extraction and translation of the data from tapes. In the case that special processing is required on the extracted data, the user can pipe the output of the generated program to a user process

* The tape type refers to the logical, not the physical, tape so that, for example, a tape issued monthly containing toll call records in the same format would be of the same type each month and require only one dictionary.

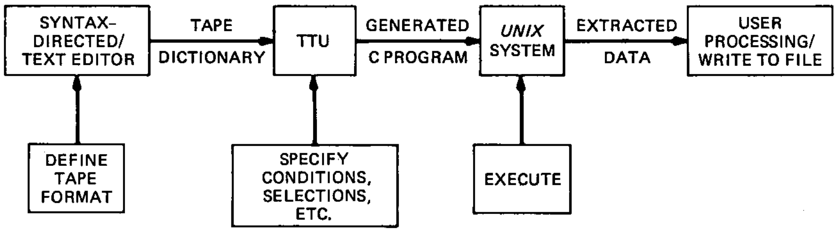
SYNTAX-DIRECTED/ TEXT EDITOR
TAPE DICTIONARY
TTU
GENERATED C PROGRAM
UNIX SYSTEM
EXTRACTED DATA
USER PROCESSING/ WRITE TO FILE
DEFINE TAPE FORMAT
SPECIFY CONDITIONS, SELECTIONS, ETC.
EXECUTE

Fig. 2—User interaction with TTU.

or specify the name of the processing subprogram to TTU at the time of making the selections. The generated program will call the processing program at the appropriate places, eliminating expensive print and read operations. Finally, the generated program is compiled and executed in the usual manner.

5.1.1 Tape dictionary format

A tape dictionary is a file containing the information about the structure of the records in that tape. The first line in the dictionary contains the file name, followed by physical block length and Variable Length Record (VLR) tape indicator (y for VLR tape, n for fixed length record tape). This is followed by a block of lines for each record type in the tape. The first line in each block contains the record name followed by the record identifier string, the identifier indicator (y if the record has an identifier and n if it does not have an identifier) and the level number (1 if there is only one record type or unrelated multiple record types on the tape; if the record types on the tape form a hierarchy, then it is the level number of the record type in that hierarchy). This first line in each block is followed by a sequence of lines, one for each field in the record. Each line for a field contains the field name followed by the field type indicator (s for alphanumeric string, n for numeric, b for binary and p for packed integer), key indicator (y if it is a key and n if not) and the field length.

5.1.2 Request format

Once a tape dictionary is created, the user invokes TTU for specifying the extraction request. TTU requests the tape name, searches its dictionaries for the requested tape dictionary and displays the record and field formats for that tape. Each field name is prefixed by a type indicator of s if the field is a string, n if it is a numeric string, or p if the field is a packed decimal, etc. The type of the field influences the format of requests for that data.

For each record, TTU requests a condition (i.e., a record selection criterion for that record). If the user wants all records of that type, a carriage return should be entered. Otherwise, the condition should be entered in the following DNF (Disjunctive Normal Form) format:

field_cond & *field_cond* &...& *field_cond* | *field_cond* &...& *field_cond* |...,

where "&" means "and" and "|" means "or".

The *field_cond* is a field name (from the list provided by TTU) followed by a relational symbol (=, <, >, <=, > =) and a value or another field name. For example, to retrieve all state records where state is "NJ" or the revenue is greater than 15,000 and revenue is greater than expenses

```
state = "NJ" | revenue> 15000 & revenue>expenses.
```

Following the record selection process, the system prompts for field selection (i.e., record projection). A program will be generated to translate and return only those fields within a tape record which are requested. In this step the fields to be returned are identified. The default assumption is that all fields used in record conditions will be returned. In the previous example, state, revenue, and expenses will be returned. The response to the selection prompt may be a list of additional fields, separated by spaces, to be included in the retrieval; `ALL`, meaning retrieve all fields of this record type, or `ALLBUT` followed by a list of fields (separated by spaces) which are not wanted. The request for data is now complete. All that remains is to let the system know where the generated program is to be stored. For the target program generation, the system will default to a KS file, an abstraction of efficient C programs to read tapes, supplied with the system. This default can be overridden in order to use other program abstractions. The final step is to compile the generated program and execute it to read a tape.

5.2 Individual components

As illustrated in Fig. 1, TTU has two basic components, a syntax/semantic analyzer producing an attributed retrieval tree based on the user request and a code generator using this tree to generate the final program from a target program abstraction. The syntax analyzer is a YACC (Yet Another Compiler Compiler) generated parser[6] for the simple DNF condition language illustrated in the previous section. The semantic analyzer builds the attributed retrieval tree. The root of the tree contains all the attributes of the tape's global characteristics, for example the block length. Below the root, the internal node structure of the attributed tree corresponds to the record structure on the tape. Thus, there is a node under the root for every level-1 record on the tape, there are level-2 nodes for the level-2 records on the tape under the corresponding level-1 nodes, and so on. These nodes are called record nodes. Every record node contains the corresponding record attributes, such as the user condition, the key string distinguishing it from other records, etc. Additionally, every record node has a set of leaf nodes for the corresponding fields in the record. The attributes of the field nodes are the items such as the field type and field length.

The code generator of TTU takes an attributed retrieval tree as input and interprets a target program abstraction given in a separate file to produce the target program equivalent to the user's request. The program abstractions are written in a special language called KS. Initially, the code generator starts at the root of the attributed tree. It reads the program abstraction one character at a time. Each character is printed to the output file unless it is one of the *meta* characters in the abstraction language. The *meta* characters are directives to the code generator for special processing to be done. The simplest of the meta characters asks for the substitution of a particular attribute value of the current node in the retrieval tree. This is similar to substitution in macro languages. There are meta characters which direct the code generator to repeatedly interpret a fragment of the program abstraction for every node in the retrieval tree whose attribute values satisfy a specified condition. In such iterative interpretations, the traversal of the retrieval tree can be limited to the subtree rooted at the current node, the path from the root to the current node or just the immediate successors of the current node. Iterative interpretations based on the attribute values of the current node are also possible.

The processing flow and the loop structures in data extraction programs seem to be isomorphic to the structure of the data in the source databases. This observation has allowed us to design and interpret the KS language for an attributed retrieval tree in such a *hand-in-glove* manner. The current version of the KS language is found to be rich enough to express the abstractions of programs in a variety of languages ranging from assembly to COBOL to C for data extraction from tapes and from IMS databases.

VI. WHAT WE HAVE LEARNED

In the process of developing the IMX/TTU system and changing it in response to user needs and suggestions, several things have become clear. The first is that, with proper tuning, fast processing of data can be done on *UNIX* systems. It is not unreasonable, for example, to think in terms of bringing large data sets to *UNIX* systems and processing them from tapes.

The program generation approach to this problem allows us to use efficiencies (e.g., loop unwinding) which would not, typically, be used by a programmer asked to do the same job. This implies, particularly in the case of TTU, that the process of generating a TTU program in C is easier than requesting one from a programmer. In addition, the generated program will probably run faster. The resulting programs have very predictable structure but are relatively long and intricate.

The flexibility provided by the table-driven program generation approach has proven very valuable in allowing us to make expert programming knowledge available to end users. As an example, in a recent exercise where TTU performed slower than expected for a class of tapes, we were able, in a couple of hours, to add several improved algorithms by allowing an expert to modify the code generation table. The mode of transferring expertise by sitting down with an expert, adding suggestions to the code generation table, trying them out and altering them in real time was very productive.

The format of data in database management systems is relatively predictable (or at least hidden from the programmer). On the other hand, the formats used on tapes can be quite idiosyncratic. Consequently, while we expect IMX to handle nearly all IMS database structures, TTU translates only a subset of tape types. The objective has been to handle tapes produced by common COBOL, Fortran and PL/I programming techniques. This includes string, numeric, packed decimal, and fixed point binary. Adding other codes, e.g., Binary Coded Decimal (BCD) and floating point binary, is under consideration.

The addition to TTU (not yet to IMX) of the option for users to add their own processing routines has proven useful in terms of both flexibility and efficiency. Although its use requires programming ability, which the simple use of the system does not, it provides a growth path to more sophisticated use of data extraction.

VII. ACKNOWLEDGMENTS

We would like to thank Bob Kayel for motivating us to work on the design idea of separating the large production environments from the medium-sized analytical systems and for the encouragement in the initial stages of this work. We recognize the contributions from Bill Shugard and Griff Smith in preparing the target program abstractions for TTU and appreciate the suggestions and feedback from various users of this system.

REFERENCES

1. *UNIX System V User Reference Manual,* Release 2.0, AT&T Bell Laboratories, Inc., December 1983.
2. G. G. Smith, private communication.
3. M. Ganapati, C. Fischer, and J. Hennesey, "Retargetable Code Generators for Compilers," ACM Comput. Surveys, *14,* No. 2 (December 1982), pp. 573–92.
4. C. M. R. Kintala, "Attributed Grammars for Query Language Translations," Proc. Second ACM Symp. Principles of Database Systems, March 21–23, 1983, pp. 137–48.

5. B. A. Bottos and C. M. R. Kintala, "Generation of Syntax Directed Editors with Text Oriented Features," B.S.T.J., *62,* No. 10 (December 1983), pp. 3205–24.
6. S. C. Johnson and M. E. Lesk, "Language Development Tools," B.S.T.J., *57,* No. 6 (July–August 1978), pp. 2155–76.

AUTHORS

David G. Belanger, B.S. (Mathematics), 1966, Union College; M.S., 1968 and Ph.D., 1971, (Mathematics), Case-Western Reserve University; Assistant, 1971–1974, Associate, 1974–1979, Professor of Mathematics and Computer Science, University of South Alabama; Computer Specialist, U. S. Army Corps of Engineers, 1973–1979; V. P., Gulf Coast Data Systems, Mobile, Alabama, 1977–1979; AT&T Bell Laboratories, 1979—. Mr. Belanger is currently Head, Advanced Software Department. His research interests include database management, automatic program generation and distributed computer workstations. Member, ACM, IEEE Computer Society.

Chandra M. R. Kintala, B.Sc. (Electrical Engineering), 1970, Regional Engineering College, Rourkela, India; M.Tech. (Electrical Engineering), 1973, Indian Institute of Technology, Kanpur, India; Ph.D. (Computer Science), 1977, Pennsylvania State University; Assistant Professor of Computer Science, University of Southern California, 1977–1980; AT&T Bell Laboratories, 1980—. Mr. Kintala is presently Supervisor, Advanced Programming Environments Group. His current research interests include programming environments and compiler techniques for application languages. He is also an Adjunct Professor of Computer Science at Stevens Institute of Technology. Member, ACM, IEEE Computer Society, Sigma Xi, Phi Kappa Phi and Who's Who in the East.