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

T—A Data Management System

By R. J. YANOFCHICK*

(Manuscript received March 1, 1985)

T is a data management system running under the *UNIX*™ operating system that provides unique facilities not commonly found in other data management systems. These powerful data manipulation facilities can access data and programs stored in *UNIX* system files, as well as data stored within a T database. T allows new structure to be added to an existing database without modification of existing data. It also allows multiple views of a database, which can be used to prevent access to privileged data by unauthorized users, as well as to provide some fairly sophisticated restructuring capabilities.

I. INTRODUCTION

T is a hierarchical data management system written in the C programming language[1] to run under the *UNIX* operating system. It was designed to experiment with strategies that would impose structure on existing data and easily modify that structure as needs arose. As a consequence, it reduces data duplication and the programming effort necessary to restructure existing data, while it allows users an appropriate level of control over and access to data. By providing powerful data manipulation and restructuring facilities, T allows users to manipulate and extract data in a form suitable for use by analytical tools available on *UNIX* systems or provided by other users; it thereby permits users to combine specialized tools to build more general and useful tools. Keeping the amount of information necessary to describe a database to a minimum and providing a more natural, understand-

* AT&T Bell Laboratories.

able table of contents form for this information makes setting up a database to be managed by T a simple 15-minute task. The query and command language provided is a simple, yet powerful, English-like language that is easy for even the unsophisticated user to understand.

T has the following unique data management features:

- Dynamic modification of views of data
- Access to data and programs stored outside the database
- Dynamic generation of data
- Query logging and modification.

Often, at least part of the source data to be used in a data analysis process already exists as one or more data sets. Rarely, however, are these data sets structured to be used without modification in a specific data analysis process. The data may be stored by a data management system in a structure not compatible with the needs of the current application. These data sets may also be stored as flat data files. In this case, chances are good that the format of the individual records does not conform to the requirements of the application. Before any real analysis begins, programs must be written that operate on these data sets to extract and reformat the data necessary to perform the analysis. The result is duplication of data. The more data are used by different applications, the more data are duplicated and programming effort is expended to extract and reformat data.

A single database managed by T supports many analytical studies, each having different data access requirements. Users of the system range from staff support personnel with little or no programming experience to analysts with experience in some programming language. Because the uses of data vary widely, and the needs of applications include performance as well as functionality, it is generally agreed that no existing data management system suits the needs of all applications. However, the data extraction and restructuring demands that are placed on such a system can be characterized well enough to provide a general framework for most analytical applications.

The characteristics of the computational demands that would be placed on such a system are not as well understood. Simple commands are provided to enable straightforward information generation. For example, these commands enable the user to input, modify, locate, retrieve, and output data. Other computational needs are not as neatly characterized; either these needs are not known ahead of time or else they are expected to be evolving. For this reason, T focuses on data retrieval, extraction, and manipulation, and on providing a method for accessing existing computational processing functions. By providing access to computational facilities outside of T, the analyst is free to choose those facilities that best satisfy the needs of the application.

For example, data on network configuration can be retrieved from a database and used to produce a graphic display of the network.

T also provides password security for individual databases; redirection of input and output; query logging, which allows users to save sessions for reuse later; concurrency control for one writer with multiple readers; and access to data and programs that are not physically part of the database.

II. SYSTEM ARCHITECTURE

T is implemented as three main modules: a file handler, a primitives module, and a language module (see Fig. 1).

2.1 The file handler

The file handler used by T is a set of subroutines that implement B-trees.[2,3] These subroutines access B-trees via a list of key-value pairs, sorted by key. This list is implemented with a prefix-compressed B-tree in which prefixes common to consecutive keys are factored out. These routines support one writer concurrently with multiple readers, preserving read consistency. This read consistency means that each user retains a consistent view of the database during a query session. That is, during a query session, only those updates made during that session are visible to the user. This read consistency can be crucial to meaningful data analysis.

Communication with the file handler is through a file handler interface. Essentially, these routines simplify the calls to the file handler and, in some cases, slightly modify the functions of some of the lower-level routines. This interface also provides a method for users to access a database directly from C programs.

2.2 The primitives module

The primitives module implements the user-level functions provided by T. These functions include data input and modification, location and retrieval of data, input/output redirection, provision of alternate views of data, and access to standard *UNIX* system functions. For example, these functions allow a user to access data from a data file and programs stored in *UNIX* system files, as well as data stored within a T database; create temporary data files from data extracted from these sources; and then use these as inputs to external programs

Fig. 1—System architecture of T.

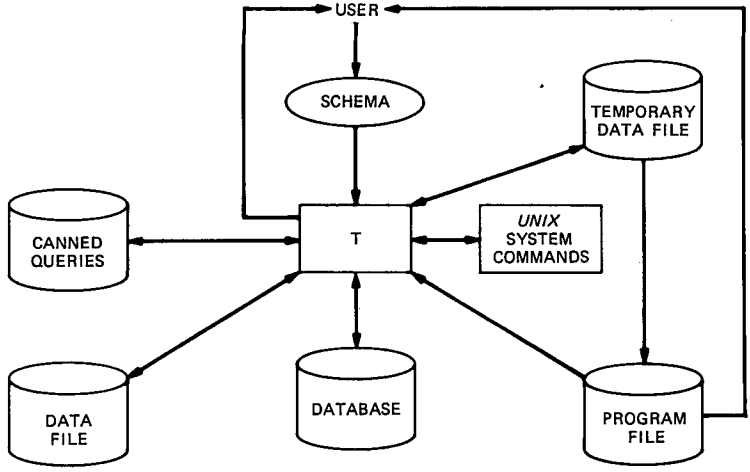

Fig. 2—Communication paths for the primitives module.

or standard *UNIX* system functions (*UNIX* system commands). Figure 2 illustrates the communication paths provided by this module.

2.3 The query language module

As currently implemented, this module accepts input from a user terminal, interprets the query command, checks syntax, and makes appropriate calls to functions in the primitives module. This module is replaceable by other query languages such as HISEL.[4] The query language currently implemented with T is English-like and fairly nonprocedural.

III. SYSTEM ATTRIBUTES

3.1 Database structure

A user-supplied schema file, which represents a hierarchy in a table of contents format, describes the structure of the database to T. A schema file contains a line for each record type (node). Each line contains the name of a node, the attributes associated with that node, and the hierarchical level of the node. The level of each node in the hierarchy is indicated by the number of tab characters preceding the node description: no tabs indicating level 1, one tab indicating level 2, etc. For example, the following schema describes a database for a sales organization:

```
market-segment (name;)
    sales-rep (name;)
        customer (name; sales; address; contact; telno;)
            product-line (name;)
                product (name; price;)
```

Fig. 3—Hierarchical structure of the schema file.

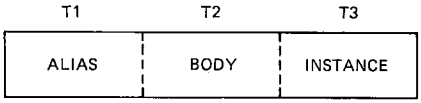

Fig. 4—Key structures.

The semicolon-separated list enclosed by parentheses identifies the fields associated with each node. These are optional. The hierarchical structure for this schema is shown in Fig. 3.

To initialize a database the user supplies the name of the database and the name of the schema file that describes its structure. During the initialization process the user is prompted for a password. This password will be requested whenever the database is accessed using this master schema. An encrypted version of the password and the name of the master schema file are stored within the database. In addition, each node (record type) is assigned a unique numerical alias that is used internally in all query processing and to link nodes when alternate views of the database are requested. Each data record in the database has a unique key associated with it. These keys are generated during data entry. Aliasing node names, which may be long, permits keys to be more compact. The structure of a key is shown in Fig. 4. To ensure key uniqueness the instance (T3) is an integer indicating the number of nodes of this type that have previously been stored in the database. The body (T2) is a concatenation of T2 and T3 from the parent key. This structure provides efficient traversal of the database. Given a node key, it is possible to locate a child of this node simply by replacing T1 with the alias of the child. A partial key search will return the first child of the type requested or indicate that no such child exists. Locating the parent of a node is similar. Replacing T1 with the alias of the parent and truncating T3 produces the key of the parent node.

3.2 Subschemas

Subschemas in T are used to invert database structure, shield data from users, and provide efficiency in retrieval. An alternate view or

subschema is an abstract model of a portion of the conceptual database or master schema. In addition to promoting logical data independence, a subschema may also provide a convenient data protection facility.[5] For example, there are situations in which the owner of a database may wish to create a subschema allowing other users access to part of the database but shielding some nodes from public access. In another case, the relationships among nodes may be different in a subschema from what they are in the master schema. For example, a market manager may wish to view the database in Fig. 3 as

```
market-segment
  product-line
    product
```

whereas a product manager may wish to view the database as

```
product-line
  product
    market-segment
```

When utilizing these subschemas neither the market manager nor the product manager has access to sales representative and customer information, since they are not contained in their conceptual views. These subschemas provide different users with their own conceptual view of the database, regardless of how the data have been stored. While other data management systems provide access to subschemas, they typically construct secondary indices by processing the entire database. Because of this processing, the database administrator, not the user, typically generates the subschema. This method also involves overhead and maintenance problems as new data get added to the database. In the approach taken by T, since the schema is separate from the actual database, and does not involve secondary indices, generating a new subschema becomes a simple mapping of one structure to another. Since this mapping does not involve accessing the actual data, no overhead is incurred, and adding new data has no effect on a subschema. In addition, generating a new subschema becomes a process available to users.

Defining a subschema for a T database is easy and does not require the assistance of a database administrator. The user simply creates a schema file that defines the new structure. A subschema may be installed when a query session is initiated or during a query session via a command provided by the query language. When subschemas are defined, there is no need to identify the fields for each node; all properties of a node are carried forward to the subschema. Some restrictions are placed on subschemas. One is that the list of nodes contained in a subschema must be a subset of those defined in the master schema. The relationships between nodes may change, but no

new node types may be defined by a subschema. Another restriction is that users employing a subschema have read-only access to the data. Without these restrictions, the original contents of the database could be corrupted. In addition, there are some query commands that are locked when using a subschema. For example, the subschema defined for a product manager is an inversion of the master schema that may result in a many-to-one relationship between products and market segments. Thus, a request to fetch the next market segment within the parent (product) may be ambiguous, since the current market segment may have several parents. In general, T attempts to make available to subschema users only those commands that have unambiguous interpretations when the subschema is mapped to the underlying database. Users of subschemas in T incur no performance penalty. Rather, in some circumstances, using subschemas can simplify queries and provide more efficient access to information. Figure 5 shows the relationship between schemas and T.

While T does not allow new nodes to be defined in a subschema, it is possible to define new nodes in the master schema at any time without restructuring the database. These new nodes may be inserted at any level of the hierarchy beneath the root (level 1). For instance, to add service-center as a child of product-line in the database shown in Fig. 5, one need only insert a description of service-center in the master schema, as shown below:

product-line (*name*;)
  service-center (*name*; *address*; *state*; *telno*;)
  product (*name*; *price*;)

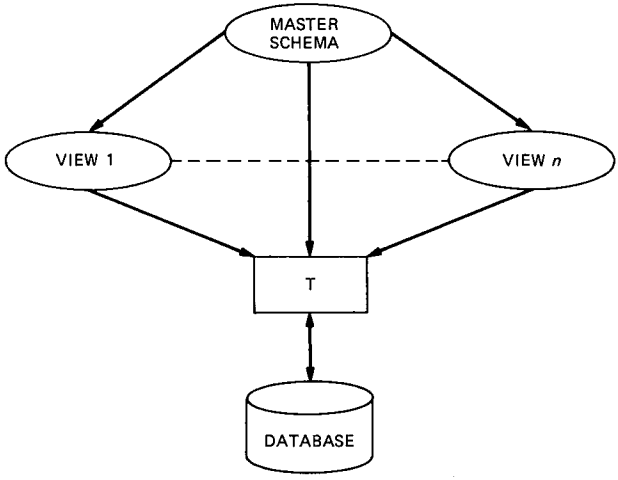

Fig. 5—The relationship between schemas and T.

3.3 Derived data

A unique and powerful feature provided by T is access to data that is not physically stored within the database itself. Data of this type are called virtual or derived data.[6] Three variants of derived data are supported by T: type 0, type 1, and type 2. Type 0 specifies that the data to be accessed either already exist in a file or will be the result of the execution of some process. In either case, the entire data set is treated as one logical record. For example, document abstracts can be stored in *UNIX* system files and accessed as part of a T database. Each abstract is treated as a single record during processing. In contrast, type 1 specifies that each line in the external file, or each line returned by some process, is to be treated as an individual instance of its node type. For instance, data of this type might be generated periodically by some external process, and rather than updating the database, it might be stored in a standard *UNIX* system file that is accessed whenever this type of node is referenced. Data of type 2 are always an executable process. Here, values are generated that do not exist in the database. A process or chain of processes to be executed occurs at retrieval time and, while the actual results do not exist, the ability to generate them does. For example, a report can be generated using data extracted with a prior query request, or the results of a query request may be displayed graphically rather than as lines of text. Using this methodology, data analysis may become a natural extension of the retrieval process. For instance, if a set of mathematical models and other analytical functions are described as data elements contained in a database, access to these functions becomes a simple retieval command, thus allowing analysis and modeling to be done without leaving the current query session. This can help to create a simple, yet extensible environment for the analyst.

Whether or not a node is actually derived is determined at run time. Nodes that could potentially be derived are identified in the schema by prefixing the node name **n*, where *n* is either 0, 1, or 2. The entry

*1 product (*name*; *price*;)

indicates that product information may be derived, and, if it is, each line is to be treated as a separate product record instance. Identifying a node as being derived in the schema file does not mean that each instance of that node actually in the database must be derived. During data entry the user specifies whether a particular instance of a node will be derived.

An interesting side effect of derived data is that the data item associated with a derived node is the name of the file to be accessed on retrieval. Since T provides interactive update for data items, a user can use this facility to dynamically change the source from which data

are retrieved. A user could, for instance, switch among several analytic models simply by interactively changing the name of the file to be executed. Since all users of a database retain a consistent view of the database during a query session, this dynamic switching does not affect other concurrent users of the database.

Accessing derived data with T is identical to accessing data actually stored in the database. There are no semantic differences or subtleties to contend with. The primary command to retrieve data with T is the `find` command. In its simplest form the command

`find` *customer*

retrieves the first customer node and displays the contents. The command

`find all` *customer*

locates all customer nodes. Instead of immediately displaying the results, T responds with the number of records retrieved. The user can then decide to display all the data retrieved, a portion of each record, or ignore the results entirely. To display output T provides the commands `print` and `fprint`. The command

`print` [*attribute list*]

where the optional attribute list contains the names of individual fields separated by spaces, will display the requested results at the user's terminal. If no attribute list is provided, the entire record is displayed. The command

`fprint` [*attribute list*] > *file*

redirects the results to *file* rather than displaying them at the user's terminal. The `fprint` command provides a data extraction capability from T databases.

Assume that we had included a node, c-report, as a part of our master schema, with the definition

*2 c-report

This node will access a report generator and produce the requested report. The set of commands

```
find all customer where sales gt 1000000
fprint name sales > foo
find c-report
```

would retrieve all customer records having sales in excess of 1000000, place the customer names and sales figures in file foo, and generate the customer report. It is assumed that the report generator being used here expects its input to be in a file named foo.

3.4 Query logging

At times the set of commands that comprise a query session need to be resubmitted periodically. This could, for example, be done to generate periodic reports using data extracted from a database. If the exact set of commands to be executed is known ahead of time, they can be entered in a file that is passed as input to the query processor. There are times, however, when the exact syntax of the commands or the proper sequence in which they should be executed is not known ahead of time. In other cases a sequence of queries pertaining to one set of data could, with slight modification, be used to retrieve a different set of data. For example, to modify the report in the example given above to select only customers with sales up to 1000000, the operator `gt` can be changed to `le` and the sequence of commands resubmitted.

During a query session, T keeps a log of those user commands that do not modify data or previous queries. The commands `retype`, `copy`, `remove`, `edit`, and `redo` provide the user the ability to manipulate previous queries and resubmit an individual query or a group of queries with a single entry. At the end of a query session, the user is given the opportunity to save a copy of this query log. Query sessions that have been saved need not be entered manually each time they are used.

IV. PERFORMANCE

Real-time response to T queries is acceptable. In timing experiments run on an AT&T 3B20S computer, under normal load—running System V, Version 2.0.2—queries involving key retrieval and pattern matching searches executed against a file containing 1.1 megabytes (10,654 records) in less than 10 seconds real time. These queries were constructed to ensure that the entire database was searched and only the last record in the file satisfied all constraints. The largest database known to have been accessed by T contained about 40 megabytes. More typical applications vary from 1 to 20 megabytes. It is extremely difficult to obtain meaningful performance figures for an interactive data management system. This is especially true of the hierarchical model. Much of the performance depends on the actual structure of the data and the type of information requested. Actual performance is also affected by the load distribution of the system at the time operations are initiated. While response time is an important factor in performance evaluation, consideration should also be given to whether the system makes efficient use of user's time. In providing unique features such as access to derived data, dynamic modification of views of data, and query logging, it is felt that T does help users in this area.

V. CONCLUSIONS

T was developed to experiment with concepts that would make data access, retrieval, and manipulation easier for certain types of database users. These concepts include (1) the ability to interactively modify user views of data; (2) the ability to access data and programs stored outside of the database, which provides a more natural interface to existing information and computational functions; and (3) the ability to access, rearrange, and modify previous query commands. Several of these have proven useful in a variety of analytical studies. The ease with which alternate views of a database can be constructed and the fact that they may be invoked dynamically during a query session have proven to be valuable assets to data analysts using the system. The concept of derived data has reduced the amount of redundant data stored on disk and provided greater flexibility by permitting access to existing computational and graphical functions. It makes access to a variety of heterogeneous capabilities natural within the same query language. Alternate input and output facilities provide simple mechanisms to access standard queries and provide a data extraction capability. Query logging provides a simple and flexible method of generating standard query procedures, as well as providing language extensibility and query reuse.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall, New Jersey, 1978.
2. P. J. Weinberger, private communication.
3. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Reading, Mass.: Addison Wesley, 1968.
4. E. R. Gansner et al., "Semantics and Correctness of Query Language Translation," Proc. 9th Principles of Programming Languages, Albuquerque, N.M., January 20, 1982.
5. J. D. Ullman, *Principles of Database Systems*, Potomac, Maryland: Computer Science Press, 1980.
6. G. Wiederhold, *Database Design*, New York, N.Y.: McGraw-Hill Computer Science Series, 1977.

AUTHOR

Raymond J. Yanofchick, B.A. (Economics), 1979, Rutgers University; M.S. (Computer Science), 1983, Stevens Institute of Technology; Bellcomm Inc., 1966–1972; AT&T Bell Laboratories, 1972—. Mr. Yanofchick has been involved with applied research into minicomputer-based data management systems. He currently works in the Marketing Analysis Systems Department, exploring software tools and environments appropriate for market analysis. Member, ACM, AAAI, IEEE.