Clifford

1

BNL--43075

DE90 003970

A C Programmer's view of a Relational Database*
Tom Clifford, Roger Katz, Cindi Griffiths
AGS Department, Brookhaven National Laboratory
Upton, New York 11973

Abstract

The AGS Distributed Control System (AGSDCS) uses a relational database (Interbase) for the storage of all data on the host system network. This includes the static data which describes the components of the of the accelerator complex, as well as data for application program setup and data records that are used in analysis. By creating a mapping of each relation in the database to a C record and providing general tools for relation (record) access, all the data in the database is available in a natural fashion (in structures) to all the C programs on any of the nodes of the control system. In this paper the correspondence between the Interbase relations and the C structure is detailed with examples of C typedefs and relation definitions. It is also shown how the relations can be put into memory and linked (related) together when fast access is needed by programs.

1. Introduction

In the design of the AGS Distributed Control System (AGSDCS) it was recognized that most of the data that is resident on the host

*Work performed under the auspices of the U.S. Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

EC 1 8 1989

OSTI

system can best be managed by a database management system (DBMS).

[1] The host resident data is divided into three classes based upon the objects the data is for. The three data classes are accelerator equipment data, program input data and accelerator readings data.

The accelerator equipment data is static data which describes physical characteristics of the accelerator complex. This includes survey information as well as the information needed for control, such as conversion functions and addressing. This data, which is static and often entered by hand fits very well into the model of data which can be successfully managed by a DBMS. The DBMS provides for easy entry of data and tools for viewing and changing data.

The setup parameters for applications programs as well as the data collected by these programs can also be a part of the host database. In this application the database is used to help with the organization and management of the data rather than the data entry. Data sets are stored in relations in the database rather than in named files in the host file system. The DBMS provides tools for adding and deleting records and changing data in a record. The DBMS provides for multiple user access to the program data for users on different nodes in the host system.

Most of the data which needs to be managed fits quite naturally into a relational model so it was easy to arrive at the need for a

relational DBMS as an integral part of the AGSDCS. The requirements for the DBMS were modest, it had to run on Apollo computers (in 1986), have an interactive sql interface and support for C in its host language interface (HLI). The Interbase database ment these requirements.

It was necessary to provide tools which would make the database data easily accessible to C programs which run on the host computers. It was felt that the HLI with its special syntax and pre compiler was an easy interface to the database for the programmers. It was felt that a few C procedures could provide the database interface for most of the controls programs. However, data needed for routine control of devices (address, limits, conversion parameters, ...) had to have an access time on the order of 1 micro second for an individual datum.

2. Relations and C structures

The key to our C use of the DBMS is the correspondence between relations and C structures. An example of this correspondence is shown in Table 1. The difference between the definition of the relation and the C type definition in Table 1 is one of notation. Without knowing the syntax of the relation definition or the C typedef a simple algorithm could be defined to build either definition from the other one.

In order to provide a uniformity for all the relations in the

database the key field in each relation, the record name, is a character string of not than more 20 characters. Thus each relation has the same data type, in the C sense, which identifies the relation record.

Because all the relations in the database have a corresponding C structure and keys all the same type it is easy to identify and write some basic generic access tools.

3. Access Tools

The following four procedures were identified as being fundamental to our needs; `get_names`, to retrieve the list of key names for a relations, `get_record`, to retrieve a named record form a specified relation, `put_relation`, to write a record into a specified relation, and `get_relation`, to retrieve all the records in a specified relation.

Table 2 shows the arguments for each of these four routines. The routines are written in the Interbase HIL. Because the database meta-data contains the type of data for each field of a relation, one routine can be used to get data from any relation and pack it into the correct C structure without reference to the C structure itself. The typedef for the C structures are not used by the database tools. As new relations are added to the database the generic routines are available to operate on the new relation without any change or recompiling of the tools. If a relation

definition is changed, the generic routines can still return the record data packed correctly into the C structure.

The routines listed in table 2 all return to the user a pointer to memory where the database data is returned. The user of these routines casts the returned data into the C structure that maps to the relation being used and is able to refer to the fields in the relation in the normal C pointer notation, `relation_name->field.`

Table 3 shows a section of code using the `get_relation` routine. Note that the relation is treated as an array of structures in a very natural fashion.

Because all the key fields in a relation are the same type, a 20 character string, the one `get_names` routine works for all relations in the database. The record names are returned as an array of strings. When new relations are defined in the database the `get_names` procedure will work without any other change to the system.

The `put_record` routine is similar to the `get_record` routine. It allows data from a C structure to be written to the database. There is a limitation to this routine because it does not allow just one field of a record to be changed, the whole record must be replaced. To effect a one field change the record must be fetched (`get_relation`), a field changed in the returned structure and the record put to the database (`put_relation`).

Because the Interbase HIL supports dynamic sql it is possible to define queries to the database at run time. This feature allowed us to write a routine to get records from a relation according to a qualifying string in sql syntax. This routine `get_record_qualified` can do anything that a sql command entry can do when working with only one relation. Its arguments are shown in Table 3. The limitation of working with only one relation at a time can be overcome at the database definition level by defining "views" which can join together relations. These tools will work with views just as they do with relations. When a view is defined, a C typedef must be defined to correspond to the view.

4. Speed and Linked Relations

The time it takes to do a `get_relation` under the best conditions is on the order of 0.01 seconds. This best case occurs when both program and data are in physical memory so there is no disk access in the routine. If code and data must be swapped into memory in order to do the data access the access time can be over an order of magnitude larger. Such results are expected for a virtual memory system with demand paging. This speed of access is comparable to the time it takes to open a file and read a record from the file and is acceptable for operations like reading a data set or displaying a menu of program parameters. It is too slow for operations involved in sending commands to devices or in the unpacking of the reports sent from devices. These operations

require many fields from about a half dozen relations. In the AGSDCS there can be many programs on many nodes sending commands and receiving reports from devices. To pass all such operations through the database would create a bottle neck that we wanted to avoid.

In order to quickly access the data in relations enumerated above we read relations into memory and link the relations together by replacing key name links with record indices. Access to fields is then done using indices into the arrays of records. Generic tools are provided for doing this linking so that, if there is need for some programs to access a set of relations quickly, the relations can be read into memory and linked at initialization time.

Consider two relations shown in table 4 as C typedefs. Relation A has a field, B_name, which points to a record in relation B. The procedure get_relation is used to read both relations into memory. The relations in memory are arrays of their respective C structures declared

```
b_record      B_record[100];  
a_record      A_record[100];
```

The generic procedure link_relations can be used to go into the array of A_records to replace the strings, B_point, with the index of that relation in the array of B_records in memory. The union notation of C provides a clean method of placing an integer into the first four bytes of a string. Once this is done, normal C notation can be used to de-reference fields in the B relation that

are related to the nth entry in the A relation as follows

```
a_b_field = B_record[A_record[n].B_name.intf].field1;
```

The trick of moving relations into memory will work only if the data in the relations is relatively static and there is some mechanism to reread the relations when the data changes. The data in the device linked relations changes only when devices are added to the accelerator so it is indeed static. Because so many programs need this device definition data the linking of relations is done once for all by a program and the linked data is written to a file that other programs can read. This file is shared by all the control programs on the host system.

5. Summary

The details of interacting with the relational database have been well hidden from the end user by making good use of the correspondence between relations and C data structures. Most of the features of the full relational database are accessible to C programs through six easy to use C procedures.

6. Acknowledgements

The evolution of the ideas expressed took place over the last few years as we came to understand better our database and the power of the C structures. We acknowledge the ideas of others who have worked on the AGSDCS.

7. References

[1] A. Stevens, T. Clifford, R. Frankel, Distribution of computer Functionality at the Brookhaven AGS, IEEE Trans. Nucl. Sci., NS-32, 2023 (1985)

Table 1 - Record and Relation Definitions

```
typedef struct {
    char      name[AGSDCS_NAME];
    char      address[2];
    short int  reset_number;
    char      channel[3];
    short int  operational_status;
    char      location[AGSDCS_NAME];
} i_combox_record_t;

NAME          varying text, length 20
ADDRESS       varying text, length 2
RESET_NUMBER  short binary
CHANNEL       varying text, length 3
OPERATIONAL_STATUS short binary
LOCATION       varying text, length 20
```

Table 2 - C Procedures Which Interact with the Database

```
get_all_record_names(relation, name_array)
    input parameter
        relation - name of the relation
    output parameter
        name_array - pointer to a list of the names of the
                    records in relation

get_record(relation, record_name, record_ptr)
    input parameters
        relation - name of the relation
        record_name - the name (key) of the record to be fetched
    output parameter
        record_ptr - pointer to the record returned from the database

get_relation(relation, records_ptr, count)
    input parameter
        relation - name of the relation
    output parameters
```

records_ptr - pointer to the records returned
 count - number of records returned

put_record(relation, data_record)
 input parameter
 relation - name of the relation
 data_record - pointer to the data to be put as a new record
 in the relation

Table 3 - C Code Using get_relations

```
stations_t      *station_ptr;
int             count, i;
get_relation("stations", &station_ptr, count, status);
for(i=0; i<count; i++) {
    display_relation("stations", &stations_ptr[i]);
}
free(station_ptr);
```

Table 4 - Structures for Linked Records

```
typedef union {
    char    name[20];
    int     intf;
} name_int_u;

typedef struct {
    char    name[20];
    int     field1;
    name_int_u B_name;
    float   location[3];
} relation_a;

typedef struct {
    char    name[20];
    float   field1;
    int     anything;
} relation_b;
```