Published in Generalized Data Management Systems and Scientific Information, OECD Nuclear Energy Agency, Paris, France, 1978

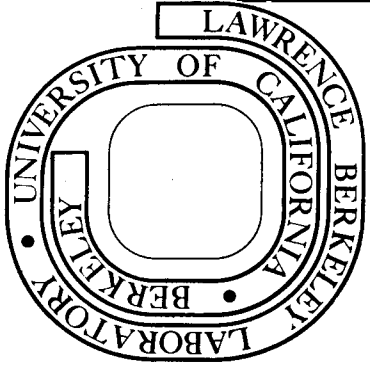
LBL-7201

AN OVERVIEW OF BDMS: THE BERKELEY DATABASE MANAGEMENT SYSTEM

David R. Richards

October 1977

Prepared for the U. S. Department of Energy
under Contract W-7405-ENG-48



LBL-7201

DISCLAIMER

AN OVERVIEW OF BDMS: THE BERKELEY DATABASE MANAGEMENT SYSTEM*

David R. Richards
Lawrence Berkeley Laboratory
University of California
Berkeley, California

ABSTRACT

The design and implementation of BDMS is outlined with emphasis on those features of particular importance to the management of complex scientific data having significant numerical content. These include binary internal representation of integer and single or double precision real data element types, ability to handle vector values of numeric data elements as well as arbitrary length character and bit strings, and the provision of an extensive set of "hooks" for the attachment of user-supplied processing routines.

INTRODUCTION

BDMS, the Berkeley Database Management System[1], is a hierarchical database management system whose design was heavily influenced by the requirements of scientific data management. It has its origin in the joint development of a Particle Physics Data System (PPDS)[2,3] by the Berkeley Particle Data Group (PDG) and the Caltech Data Compilation Group.

The original conception of the PPDS software called for a separate system to manage each of the databases: a document system, a reaction data system, a particle properties system, a vocabulary control system, etc. In fact, by mid-1973 a prototype of each of these systems had been developed. It rapidly became clear, however, that the task of modifying, extending, and even maintaining these specialized systems was beyond available resources. There was ample reason to expect that the requirements for these systems would never stop changing; the nature of particle physics data had changed significantly over the past decade as the field evolved and was expected to continue changing.

---

*This work was done with support from the U. S. Department of Energy.

Searching for a different approach, we realized that it should be possible to use a single database management system that would be general enough to manage all the databases and serve as the basis for the specialized software that would still be necessary. The broad spectrum of size, growth rate, volatility, and complexity encompassed by these databases and the complexity of their intended uses demanded capabilities not provided by any database management system then in existence that could be run on our hardware (CDC 6400, 6600, 7600). Thus, we were led to develop our own system.

We are now painfully aware of the very large commitment of time and resources necessary for a project of this magnitude. In fact, our naivete at that time was probably a major factor contributing to the eventual attainment of our goals, since if we had realized how difficult it would be, we might never have embarked on the project, instead simply reducing our expectations.

There is, on the other hand, a clear advantage in local control of system development. When some new capability is needed, it is possible to make a decision as to whether it is a general facility that is best built into the database system once and for all, or whether it deserves only to be put into the application software. With a system supplied by a commercial vendor, some very complicated and unattractive kludges may be necessary to add a capability, since the only option is building it outside the existing system.

In the rest of this paper, the structure of a BDMS database will be defined and the facilities of the system (version 2.1) outlined. Some comments will be made concerning those aspects of the implementation that might affect its suitability for a proposed application. Finally, our plans for future development will be outlined.

STRUCTURE OF A BDMS DATABASE

A BDMS database is structured into records, the units in which data pass between the system and disk storage. Normally, a record will have some significance to the user, e.g. a record in a bibliographic database would be a description of a single document, but this is not always necessary or desirable.

The individual data items within a record are called data elements; they are the smallest units of data with any meaning to the system, although an individual data element might have some internal structure known to an application program. A data element has a unique name and is normally referenced by name (or a synonym). There is essentially no limit on the number of data elements that may be defined for a database.

A hierarchical structure may be imposed on the information within a record when a database is defined. This means that some data elements are declared to be subordinate to other, parent, data elements. Those data elements for which no parent is declared are called record-level data elements; it is often useful to consider the record itself to be their parent. There is essentially no limit on the number of levels which may be defined in the record structure.

Within a given record, each record-level data element may occur once, several times, or not at all. Likewise, each occurrence of a data element at any level in the hierarchy may have linked to it one, several, or no occurrences of each of its subordinate data elements. There is essentially no limit on the number of times any data element may occur in a single record.

Data elements are classified into six types according to the values they can assume. INTEGER, REAL, DOUBLE, CHAR, BIT, and NODE. Integer or real (floating point) data elements may be scalars (single numbers) or arbitrary length vectors (i.e. an ordered set of numbers, which are the components of the vector). Real data elements may be single or double precision. Character or bit strings may be of any length with no limit beyond that imposed by run-time memory restrictions. Any of the foregoing data element types may occur with a null value. Pure node data elements carry no value; they may be used to link together subordinate data elements in the record hierarchy or as flags.

Any data element, regardless of type, may serve as a node in the hierarchical record structure. In general, if one of a group of related data elements may occur only once in each occurrence of the group, it should be made the parent of the rest of the data elements. However, if all of the data elements in such a group may occur multiply, they all must be linked to a pure node parent, since no single occurrence of any one of them can serve as the parent of the rest.

Any data element may be declared to be a record key. The system will then maintain an index for that data element to allow efficient retrieval. In an index, key values have a fixed length that is declared in the database definition; data element values are truncated or padded as necessary when they are put into an index.

It is possible to declare a data element to be virtual, which means that it will be recognized in input data and appear in the record buffer, accessible to user-supplied processing routines, but will be discarded when the record is stored in the database. This is useful if the input contains data elements that one does not wish to store in the database at all. One might also wish to store some data element values only in the indices, where they point to the record, without permanently allocating space for them in the record itself. This is often the case if a key value is constructed from the values of one or more data elements by a user-supplied processing routine (discussed below). In this case, a virtual data element is defined and is further specified to be a key. It is then used to hold the key values temporarily; when the record is stored, these values will be indexed before the virtual data element is discarded.

The system assigns a record ID to each record as it is created. This guarantees that each record has a unique identifier by which it can be selectively retrieved even if none of its data elements is defined to be a key so that no indices are maintained. The ID is displayed whenever a record is listed by the system.

BDMS FACILITIES

BDMS comprises a database definition compiler, a database executive, and several utility programs.

The database definition compiler is used to create a new database. It accepts a description of the logical record structure expressed in a database definition language (DDL) and generates tables describing the database. These tables then drive the rest of the system when that database is in use.

The database executive is a self-contained system providing a user interface to database maintenance and retrieval facilities through a high-level language. It has two major subsystems: an editor and a query language processor.

The editor permits a user to enter data into a new record or modify an existing one by appending, inserting, replacing, or deleting data element occurrences by means of free-format editing commands. A string substitution facility is provided for modifying the values of character string data elements. Any change to the database made via the executive is immediate effective and reflected in the indices.

The BDMS query language permits a user to search a database for those records satisfying an arbitrarily complex condition on key (indexed) data element values. The condition is constructred as a Boolean combination of key value specifications, including inequalities and ranges. Furthermore, it is possible to search for records having an occurrence of a specified data element regardless of value, or for those having an occurrence of the data element with a null value. Truncated value specification for character string keys may be used to search for those records having an occurrence of the data element beginning in a particular way. A particular record ID or range of record IDs may be included in a query explicity. The result of a query is the set of records that satisfy it. Any of them may be listed at the user's terminal, printed, dumped, modified, or deleted. Existing sets may be combined with other sets and still further conditions through the query language.

The utility programs provide for efficient initial database loading, full database dump, data file garbage collection, and rebuilding of indices for more efficient query processing and disk space utilization.

Except for the utilities, which are batch programs, the system may be used in either batch or interactive mode by simply linking it to the proper set of low-level I/O routines. The user language is identical in either mode. BDMS also may be invoked procedurally from a user-written FORTRAN program by means of a small and carefully chosen set of subroutine calls.

The database executive incorporates exits to user-supplied processing routines to allow input data validation, and data transformation on input, output, during the creation of index entries, and the processing of queries. In addition,the user may supply routines that are called just before a record is stored or just after it is fetched from the disk. The store processor routine may perform more complex data validation involving correlation of several data element values and may generate additional data element values, e.g., keyed virtual data elements as discussed above. The fetch processor routine is primarily useful to rematerialize virtual data elements when it is desirable to make them visible to a user or application program.

These "subroutine hooks" provide a powerful facility for tailoring the database executive for specific databases and applications without requiring the user to write a completely new "front end." They are particularly important to the management of scientific databases since even input data validation is likely to be too complex to be handled by the simple facilities (e.g. within a specified range or explicit list of values) typically provided by a commercially-oriented database management system.

IMPLEMENTATION

A BDMS database is divided into three system disk files: the data file, which contains the database definition and data records, the directory file, which contains the physical storage addresses of data records, and the inversion file, on which reside indices for key data elements.

On the disk, the hierarchical structure of a data record is represented by unidirectional pointers linking together the data element occurrences. There is no storage overhead associated with data elements that do not occur at all, either in the record or a particular occurrence of their parent. When a record is brought into the record buffer, it is restructured to facilitate access--the pointers are made bidirectional and relocated relative to the start of the work area, and entry pointers are allocated for all missing data elements to allow insertion. When the record is converted back into its disk-resident form prior to storage, all garbage resulting from updating activity is automatically eliminated.

If a modified record is no longer than it was prior to modification, it is stored back in its original location on the data file and any unused space following it is flagged as deleted for the data file garbage collection utility. If the modified record is longer than it was, it is written at the end of the data file, its directory file entry is updated, and the original form of the record is flagged as deleted.

Numeric data element values are stored in the database in the internal binary format of the machine being used. This decision was based upon the assumption that much of the use of ascientifically-oriented database management system involves access to the data by analysis software requiring numeric values in internal binary form. They must be converted into this form from their character representation only when initially input rather than every time they are read by an analysis program, as would be the case if they were stored in character form.

A single access to the directory file, with an entry address calculated from a record ID, provides the disk address and length of that record on the data file. This level of indirection was provided so that it is not necessary to update all index entries for a record that has been made larger by an update and has to be relocated; the only index entries that need to be changed are those for data elements whose values were actually modified. Likewise, when records are moved by the data file garbage collection utility, only their directory file entries need to be changed; all index entries remain correct.

The BKY operating system, under which BDMS is run at LBL, makes no provision for re-entrant code or updatable shared disk files, so BDMS was built as a single user system and does not support updatable shared databases. However, BKY does allow shared access to a read-only ("public") disk file, so multiple users, each with his own copy of the database system, can retrieve information from a common disk-resident database.

Likewise, BKY does not support permanent disk files, so elaborate crash recovery machinery in BDMS was not deemed necessary. Normally, a database is stored on tape, staged to a disk for use, and then staged back to another tape if it has been modified. One can then "roll back" to a previous stage of the database provided that a sufficient number of tapes is used in the storage cycle.

The only situation in which a system crash can be a major annoyance is when an interactive user has made extensive changes to a database immediately before the crash. To save such a user from having to re-key all those changes, the system records all user input on an audit file. If this file is maintained in such a way that it survives the crash (tape would be virtually foolproof), it can then be processed as batch input against the original version of the database. The audit file can also serve as a record of update activity if it is preserved as a part of normal operating procedure.

Since each user either uses a read-only public file or has his own copy of the database, we did not feel it was necessary to design an elaborate security mechanism to prevent unauthorized update. We are considering the addition of a password scheme, access control at the data element level, and possible even selective encryption to ensure privacy of sensitive information.

One of the design goals was easy transportibility. This was achieved by a) writing the major part of the system in a relatively machine-independent subset of FORTRAN IV, and b) careful modularization to isolate machine and operating system dependence in a few low-level interface routines that can be recoded easily for another system. Versions of BDMS exist now for CDC 6600 and 7600 computer running BKY, and IBM 360 series machines. An experimental 7600 version has been run under SCOPE and we are currently working on a PDP-11 version for RSX and IAS systems.

FUTURE PLANS

Further areas for development of BDMS fall into two major areas: extension to allow multiple record types ("multi-file databases"), and enhancement of the query facility to handle intra-record and non-key record types will be the ability to modify easily the definition of an existing database. The ability to formulate queries involving intra-record qualification will allow retrieval conditions to be specified in terms of the relative position of data element occurrences in the record hierarchy. We are also considering the addition of multidimensional array data element types as well as the access protection machinery mentioned in the previous section.

As BDMS has evolved, many people have contributed ideas and assisted with programming. They include Tricia Coffeen, Paul Chan, Geoffrey Fox, Marge Hutchinson, Silvia Sorell, Paul Stevens, Gill Ringland, Alan Rittenberg, Deane Merrill, Tom Lasinski, Tom Trippe, Vicky White, and George Yost. Over the course of development, support has been provided by the National Science Foundation and the National Bureau of Standards, in addition to that of the U. S. Department of Energy (in its previous incarnations as the U. S. Atomic Energy Commission and later the U. S. Energy Research and Development Administration).

REFERENCES

1. Richards, D. R., BDMS User's Manual, LBL-4683 (Revision 1); BDMS Programmer's Manual, LBL-4684, BDMS Implementation Manual, LBL-4685.

2. Berkeley Particle Data Group, Particle Physics Data System Document File, PDG-3100, Particle Physics Data System Reaction-Data File, PDG-3200.

3. Stevens, P. R., and Rittenberg, A., The Particle Data Group: Using a BDMS to Solve Data Handling Problems in Particle Physics, CALT-68-622, contribution to this study.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720