ON THE DEVELOPMENT OF DATA BASE EDITIONS*

Robert W. Taylor
David W. Stemple

COINS Technical Report 74A-2

March 1974

*A paper presented at the IFIP TC-2 Working Conference on Data Base Management Systems, Cargese, Corsica, April 1 - 5, 1974.

ON THE DEVELOPMENT OF DATA BASE EDITIONS

Robert W. Taylor and David W. Stemple
Department of Computer and Information Science
University of Massachusetts, Amherst, Massachusetts

1. INTRODUCTION

A recognized fact of life when dealing with data bases is that they evolve over time. This evolution takes place on a number of levels of data structure. The first of these, and the level with which most existing systems deal well, is the level of data occurrences. That is, new occurrences of data are inserted, and existing occurrences are changed or deleted. But through all of this, the data base structure--the schema*--remains unchanged. New item, record and set types are never added to or deleted from the data base schema, nor are the definitions of which occurrences of records are linked to which other occurrences of records, e.g. the SET OCCURRENCE SELECTION clause of the set member sub-entry, ever changed.

The static nature of the data base schema has a number of implications for the run-time modules of the data management system. Generally, it means that the storage format of records is bound at compile time for any program accessing the data base. That is, if two records A and B are related by set S, then whatever method is used for representing occurrences of this set--say chaining--there is no question concerning "where in an occurrence of record A is the pointer for following set S". The answer is always the same for record occurrences of type A. A similar argument holds for items and groups within occurrences of record type A. This is not to say that the run time modules of the data management system may not have to search through the stored version of the schema to discover the position--it is merely to say that there is a binding between record type and storage format, which may indeed be defined by the stored schema.

Because of the fixed nature of data base schemas, usually the only way to allow a data base to evolve is to dump the data base under one schema and reload it under a different one. The introduction of new record types, so long as they are not heavily linked to existing record types, is an exception to this. But the fact remains that not much flexibility beyond the introduction of new record types is allowed.

In addition to the dumping and reloading of at least a part of the data base, there is the concurrent problem of recompiling any programs which deal with the changed record or set types. The user working areas, used for communication with the data base manager, will have been given a specific storage layout by the compiler. A re-compilation and possibly a change to the program's data division may be necessary depending on the sophistication of the sub-schema facility provided by the data management system.

*For purposes of the exposition, we use the terminology of the DBTG report [1]. The issues discussed here are independent of any particular data management system, however.

This paper explores techniques which will allow data base schemas to evolve over time without the necessity of dumping and reloading or of re-compiling application programs. In particular, we explore the problem of allowing data base "editions"--an edition of a data base is defined as follows:

> The original schema is the edition O schema. This schema (compiled into schema tables), together with data occurrences conforming to it, comprise an edition O data base. An edition n + 1 schema is created from an edition n schema by one or more of the following:
>
> 1. The addition and/or deletion of one or more items within one or more record types.
> 2. The addition and/or deletion of one or more record types.
> 3. The addition and/or deletion of one or more set types.
> 4. Changes in set occurrence selection or virtual/actual result specifications.
>
> Further, an edition n schema specifies the mapping, if needed, of all editions j ($j < n$) data occurrences into forms suitable for presentation to edition n programs, i.e. programs compiled with an edition n schema*.
>
> An edition n data base is an edition n schema (compiled into schema tables) together with data occurrences conforming to editions O through n of the schema.

We further assume that an edition n program wishes to access data occurrences from both prior and later editions. We focus our discussion on the design issues which arise from such an environment, namely:

1. What features of a data base management system are necessary to support editions?
2. How can one edition differ from another and still be processed by programs compiled later or earlier?
3. What attributes of the data base are still resolvable at program compile time, implying little loss in run time efficiency? When is a re-interpretation necessary?

2. NECESSARY FEATURES TO SUPPORT EDITIONS

Consider the case where items are added to a given record type in a new data base edition. For example, record type R, edition O (R.O) may contain items A,

*Strictly speaking, it is only necessary to specify mappings from edition n-1 to edition n, but we allow the more general case for efficiency.

B, and C whereas record type R, edition 1 (R. 1) contains items A, B, C, and D. If we wish to allow edition O programs to access edition 1 records, there is clearly no problem, since record type R to edition O programs contains only items A, B, and C. Thus the system need only deliver items A, B, and C in the formats they had at data base edition O, and the program will execute properly (we explore the problem of storing new data occurrences in later paragraphs).

Retrieval of edition O record occurrences (of type R) by edition 1 programs is somewhat more complicated. Clearly a value must be provided for item D. We thus arrive at a following necessary condition in a data definition language which allows for data base editions:

> If in a later edition, an item is added to a record type, the definition in the later edition must provide a way of defining the value for accessing records of a previous edition. A data base procedure incorporated into a VIRTUAL RESULT specification will suffice for this task.

It should be clear that as soon as a generalized virtual result facility is available, the deletion of items in new editions becomes possible. In this case, when an edition O program accesses an edition 1 record, the virtual result will have to be computed since the item will be included in edition O of the record, but not in edition 1. If an item is deleted from a record type in a subsequent edition, then the edition definition must provide a means for computing the value when the record is accessed by programs compiled under a previous edition. Again, a generalized VIRTUAL RESULT mechanism will suffice.

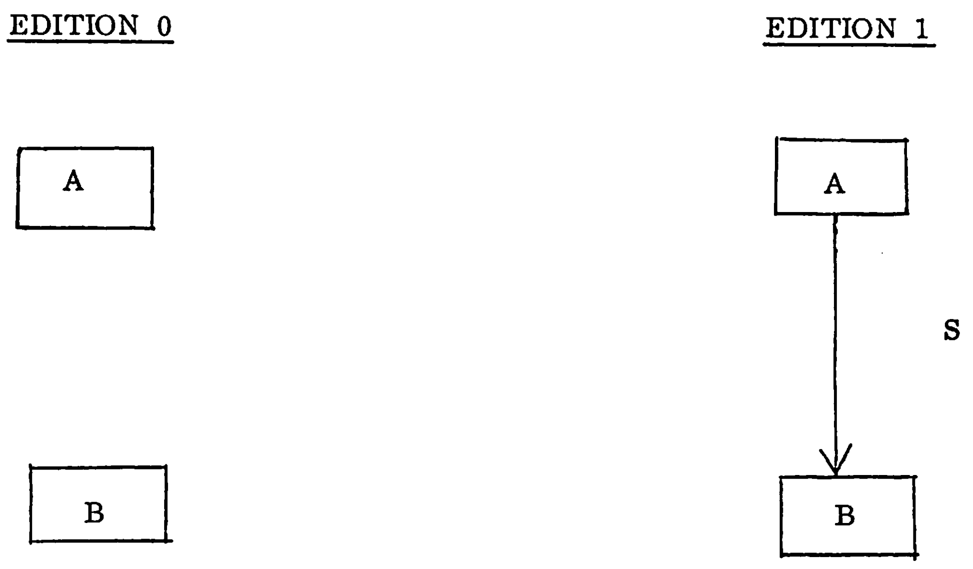
The case of accessing set definitions which may or may not be present with respect to a particular edition of a data base is similar to item addition and deletion. However, allowing set types to be added/deleted across editions imposes some limitations on the structure of the records participating in the sets. For example, suppose that an edition O record occurrence is the current one of a particular run unit, and suppose also that in edition 1 of the data base, this record type is the owner record in a set type S that did not exist in edition O (see figure 1). Assume further that the run unit is an edition 1 program and wishes to traverse members of set S. The set occurrence can be traversed so long as in edition 1 there is a proper "set occurrence selection" clause for the new set. This set occurrence selection clause must be phrased in terms of values existing in both editions. It must specify criteria whereby a record occurrence can be implicitly associated with a particular set occurrence by virtue of the value of certain items existing within the record itself. Furthermore, the data management system must have an independent access path to record occurrences specified in the "set occurrence selection" clause.

Consider the example of Figure 1. As shown, in edition O of the data base, record types A and B exist, but are not explicitly related via a set. In edition 1, the two record types are related by set S. Further, the conditions for an occurrence of record B to be considered related to an occurrence of record A (i.e., a member of set S) is that they have matching key values. This is stated in the revised syntax of the forthcoming DDLC Journal of Development [2]. If an edition 1 run unit issued the DML command

FIND NEXT MEMBER OF S SET

and the "current" record of set S was an occurrence of record type A, edition O, the data base system would locate (by some means) those occurrences of record type B whose designated item values matched those in the record A occurrence. The record occurrence presented to the run unit would be that occurrence which satisfied the edition 1 SET OCCURRENCE SELECTION clause and was first under any set ordering defined in edition 1 for set S. We can thus draw the following conclusions:

> So long as the determination of set occurrence selection is based on data values existing in the participating records (as opposed, say, to currency), and the data management system has an independent access path to the participating records, new set types can be incorporated across editions.



Sample Set Occurrence Selection Clause in Edition 1 of the Data Base.

SET OCCURRENCE SELECTION IS THRU S OWNER
IDENTIFIED BY CALC-KEY OF A EQUAL TO KEY
IN RECORD-B.

FIGURE 1

ACCESSING IMPLICIT SETS ACROSS DATA BASE EDITIONS

We now consider how this "independent access path" might be specified. The most obvious method is to make each record type a member of a singular set (i.e. one owned by the SYSTEM). This will always provide an exhaustive access path, but it also forces the data management software through a lengthy search in a large data base. A more sophisticated mechanism makes use of data base procedure mechanisms, often associated with location mode CALC. Given certain data values in the occurrence of record type A of our example, the data management system could use the appropriate values as inputs to a data base procedure which was much more than a CALC routine. Rather, this data base procedure could be a whole access method, as long as it eventually returns a data base key. Thus, by giving record type B a CALC location mode, a whole range of possible set traversal mechanisms is possible.

The most important point, however, is that the evolution of sets across data base editions requires a "matching value" philosophy when using the set occurrence selection clause. This reflects an approach similar to the match condition definition facility in the L-string of the DIAM model [3].

There remains the question of when to introduce a new record type as opposed to a new edition of an existing record type. A moment's reflection should convince the reader that the following outline should guide a data base administrator when making this decision:

A new edition is justified if:

1. An edition O program need ever access data values in a record stored by an edition 1 program

2. An edition 1 program needs to access an edition O record as if it were the augmented (or diminished) record under consideration.

If neither of the above conditions obtain, then introduction of a new record type (possibly containing as VIRTUAL RESULTS items from record types defined under the old edition) is a proper evolution of the data base. This will guarantee that the old program can never make reference to the new record type, since its name is not in the edition O program's name space.

The question of whether to create a new record type or new edition of an existing record type can also be approached using the concepts of the Entity Set Model and DIAM [3].

We first view a record type in the DBTG sense to be an A-string corresponding to an entity description (in DIAM terminology). If a data administrator is creating a new A-string (record type) corresponding to a new entity description, then the new edition of the schema should contain a new record type. The old program should be unaffected since it deals with entity descriptions which have not been revised; the old program need have no knowledge of the new entity description. If, on the other hand, the data administrator is augmenting or otherwise changing an existing entity description, then the new edition should contain a revised definition of the existing A-string, reflecting the fact that the entity level model is changed, but in a different sense.

For storing records, we adopt the guideline that creation of new record occurrences or modification of existing ones will be made using the current edition. This has several advantages. Primarily, it means that the run time modules of the data management system do not have to search back through the stored version of the several schemas in order to discover the proper storage format. They always use the most recent one, which is accessed frequently anyway and is apt to be in main storage. They do have to use the old schema in order to decode the data occurrences, but as will be seen in section 3, this does not imply a heavy overhead. In addition, the purpose of editions is to allow the data base to evolve. By always writing under the current edition, the data base occurrences will tend to be brought up to date as they are accessed. Only those records which are truly archival in the sense that they are read and never modified will remain under the old storage format.

There are some difficulties with this approach, however. In particular, when writing a record occurrence where an item or set type which existed in edition O does not exist (explicitly) in edition 1, information may be lost. This loss results from the fact that the set was explicitly represented in edition O. There is no guarantee that the data values which implicitly represent set membership are present in the new edition (after all, the set is presumably no longer of interest). This loss, however, is due to a choice made by the data administrator and should therefore be a loss of little consequence. As a practical matter, it is our experience that additional set and item types are more frequent by far than the deletion of item and set types across editions. Thus, bringing the record occurrences "up to date" will usually not imply an information loss from previous editions.

3. IMPLICATIONS OF EDITIONS ON RUN-TIME EFFICIENCY

The run-time inefficiencies of data management systems are well known. These inefficiencies are generally the result of a large amount of interpretation by the run-time modules of the data management system; they can generally be justified by savings in other areas--increased data independence, reduced program development time, etc. On the other hand, one would hope that inefficiencies would be avoided wherever possible. By this we mean that when a decision can be bound without loss of data independence it will be bound. This approach is preferable to total interpretation by the run time modules, especially if data management systems are to service applications which need a fast response.

There are several cases where a binding is possible, if desired, when dealing with data base editions.

Consider the case where an edition n program wishes to access edition m records, $n > m$. Most data management systems require that the run unit must first "register" its existence with the run-time modules of the data management system. That is, the operating system job scheduler will start execution of a run-unit, but this run-unit must still inform the data management system of its existence in order to establish communications areas in storage and to allow the data management system to resolve concurrent requests for data resources, free deadlock conditions, etc. During this "registration", the run unit could provide all information pertaining to the mechanism for transforming data occurrences from prior editions into edition n format. This is true because at program compile time all editions of the stored data base schema were (presumably) available to the compiler. Each of these editions specified how to transform earlier editions into later editions. Thus this information can be carried with the compiled version of the program and "registered" with the data management system. There is no necessity for the data management system during data access to also access the stored schema in order to find out "how can I transform edition i to edition j". That question is resolvable at compile time for all data base editions less than or equal to the program edition.

If the data base record is of a later edition than the program edition, there is, of course, no way the program can tell the system how to do the transformation. When the system discovers an edition number in the accessed record which is greater than the edition number of the accessing program, it will have to search for the appropriate version of the stored schema. Thus the stored schema is interpreted only when absolutely necessary.

4. CONCLUSION

This paper has introduced the concept of a data base edition and explored ways in which it could be used to allow data bases to evolve over time without the necessity of dumping and reloading. Of course, periodic dumping and reloading to bring all data occurrences up to the present edition, as well as recompilation of frequently used programs, may be desirable for the sake of efficiency. However, with the ability to define editions, the decision of when to bring the data base up to date can be made using a cost/benefit analysis in the same way that periodic "garbage collections" of a data base can be justified on an operations research basis [4].

For completeness, we note that there is a third kind of data base evolution, more complex than creation/deletion of data occurrences and less complex than full schema editions. This intermediate level deals with changing storage structures for a fixed data base schema. An example would be a change in set implementation specification from a mode of CHAIN to one of POINTER ARRAY with no other change to the schema edition. Another would be a change in item representation from binary to decimal.

It is felt that the techniques developed in this paper will aid in allowing this intermediate level of editions--indeed, the concept of a sub-schema already incorporates many of these flexibilities.

In summary, if we accept the evolution of data base structures--both logical and physical--as a way of life, it follows that the data base administrator must be allowed to accommodate this evolution without dumping and reloading for every change. The concept of a data base edition gives much of the machinery to let this happen without massive recompilations. It is also possible under a data base management system with editions to try some experiments which improve efficiency. As with so many programming systems, analytical and simulation techniques can often only suggest possibilities. To see if a change will really help, the only way may be to try it. If it works out, then a more complete restructuring can be undertaken; if not, then the occurrences will eventually be removed by reversing the effect of the experiment in the next edition.

REFERENCES

1. CODASYL Data Base Task Group, April, 1971, Report, Available from ACM.

2. CODASYL Data Definition Language Committee, Journal of Development, in press.

3. Senko, M. E., E. B. Altman, M. M. Astrahan, P. L. Fehder, Data Structures and accessing in data base systems, IBM Systems Journal, 12:1, 1973.

4. Schneiderman, B., Optimum Data Base Reorganization Points, Communications of ACM, 16:6, June, 1973.