INTRODUCTION

The scope of the conversion problem

There are two factors that make conversion necessary: changes in users' functional requirements and changes in their performance requirements. These changes may necessitate the acquisition of new software and hardware, which in turn, may require changes to the existing data programs. For example, the acquisition of a database management system to replace a file management system requires the integration of the original files into a database system and modification of the programs to interact with it. The replacement of a current DBMS with a new database system to provide additional functionality may require a different way of logically structuring the information (its data model) and a different kind of language interface. The establishment of a communication network between differing systems to implement data sharing will require dynamic (i.e., in real time) conversion of data between different nodes in the sense that the same item will repeatedly undergo conversion as it is needed, or alternatively, it requires conversion of programs when they travel to different nodes to access data there. Even when the acquisition of new software or hardware is not warranted, changes in a users' functional and performance requirements can require data and program conversion.

What makes conversion difficult is the proliferation of data models and levels and styles of DBMS interfaces, internal data representation, and hardware architecture. In this section of the report we will examine the technology that has been developed to perform conversions, analyze the areas which require new or improved techniques, and consider strategies for minimizing the need to convert as well as for streamlining those conversions which must be undertaken. Over the past six years, research and development has primarily centered on the problem of converting in non-dynamic environments. The first part of the second section surveys tools and technology currently available for the conversion of data organization. More recently database program conversion—probably the most difficult part of the conversion problem—has received attention. It is reviewed in the second part of the second section, which considers the research directions being taken by current research. The final component of the second section analyzes the status of the entire technology. The third section explores the factors affecting conversion, the approaches for reducing the need to convert data and applications programs, and the impact of new software and hardware technologies on conversion. The fourth section summarizes the trends in conversion needs and tools.

* This is a report of the Conversion Technology Panel which met under the aegis of the ACM/NBS Workshop, "Database Directions—The Conversion Problem". A complete report covering the other panels—Management Objectives, User Experience, and Standardization—will be forthcoming from the National Bureau of Standards.
Components of the conversion process

When conversion is necessary, users will be required to extract data from their source environment and restructure them to the form required for the target environment. While the extraction and restructuring may themselves be complex, these processes are frequently complicated further by the undisciplined nature of the source data. For example, data may exist in duplicate or contain numerous errors and multiple inconsistencies. The whole process of extracting it from its source, "cleaning the data," restructuring them to a desired form, and loading them into the targeted environment is generally referred to as data conversion or translation.

After a data conversion, particularly one involving extensive restructuring, the application programs which process the original data may not run correctly against the new data. If the amount of restructuring has been small, only a simple modification may be required, while extensive rewrite or redesign may be required when restructuring has been extensive. The process of modifying the programs to process the restructured data is referred to as application program conversion or translation. (In this report we do not consider the problem of converting programs not related to a change in data structure.) Let us discuss data conversion and application program conversion a bit further.

Concepts of data conversion

In brief and conceptual terms, the data translation or conversion process can be represented diagrammatically in Figure 1. As shown in this diagram a data translation system generally requires three components: a reader, a restructurer, and a writer. While the capability of each component depends on the individual design of a data translation system, the function of the reader in general is to access data from its source environment to prepare it for further processing. The accomplishment of this process unquestionably depends on the description of the data. Thus a data description language is needed for this purpose. The writer is the functional inverse of the reader, its function being to put the transformed data into the target environment. It too requires a description of the data structure and shares with the reader the need of a data description language. The function of the restructurer is quite different from that of the other two components. This component in general is responsible for the extraction of data from its source or internal forms and restructuring it to a desired format or structure. Usually a translation description language is required to effect this process.

In a data conversion system where limited application is intended, the three components may not be distinguishable, nor is the need of the two languages clear. For example, if one wishes to create a conversion system merely to translate EBCDIC characters into ASCII, one can create a simple system with one component and a simple data description language embedded in it. However, if development of a broadly applicable data translation system is the goal, it is clear that one must have a reader and writer capable of accessing and creating data in all kinds of environments and a powerful restructurer capable of all manner of manipulating data and of creating some data as well. Implicit in this application is the need for versatile data translation description languages.

Concepts of database program conversion

Figure 2 represents a general approach to database program conversion or translation. To convert a program it is necessary to determine the functions of the program, and its semantics. Programmers making assumptions about the state of the data may not and currently need not state these assumptions explicitly in their programs. Therefore, it will usually be necessary to provide more information about the semantics of the program than can be extracted from the program text and its documentation. Also needed in the program conversion process is information about the data structure the program originally ran against, the new structure it must run against, and how the two relate. These descriptions could be the same as those used to drive the data translation process. The program, the description of its semantics, and the description of the data conversion are inputs to the program conversion process. It uses them to determine the data originally accessed and how to accomplish the same access in the new structure, and it produces a new program to do this. Currently, the conversion process is accomplished through a combination of manual translation, emulation, and bridge programs. An automatic or semi-automatic program conversion technology is only in the early stages of research.

CONVERSION TECHNOLOGY

In the previous section, we discussed the general concepts of data and program conversion. In this section we shall discuss the technology by which a conversion is achieved.
Since most of the conversion results have been achieved in data conversion, we focus on this aspect and limit our discussion on program conversion to those cases which are the result of a data conversion.

**Data conversion technology**

Currently, the common approach to data conversion is to develop customized programs for each transfer of data from one environment to another. This approach is inherently expensive; the programs are developed for use only once and their development costs cannot be amortized. Further, the reliability is poor as one is likely to make errors as data is passed from program to program. For a large data conversion effort, this latter effect can indeed be quite unmanageable, as the tracing to the source of an error in the data at a particular point can be extremely complex. An alternative is to search for a broader approach for data conversion with a generalized system. We shall now describe such systems in more detail.

**Problem discussion**

Data exist in many varied and complex forms; Figure 3, an expanded form of the diagram in Figure 1, indicates some of the transformations that need to take place in a data conversion. This diagram illustrates the capabilities needed by the read and write process in a data conversion system. The purpose of the unloading of data from its source is to reduce the complex physical structure of the source database to a very simple physical structure, namely a sequential data stream. The source database contains not only the information that interests the user, but also a large amount of control information that is specific to a particular system. This control information is used by the system for such data management functions as overflow chaining, index maintenance, and record blocking. In many systems, when a record is to be deleted it is marked or flagged but not actually deleted. During the unload transformation, all of this system specific information is deleted. Another factor that causes complexity in the unloading process is the frequent use of pointers in a source system. Pointers are used for two basic purposes: (1) to represent relationships that exist between record instances and (2) to implement alternative access paths to the data. During the unload transformation, the second class of pointers may be discarded without loss of information. The first class of pointers, however, maintains information that must be preserved during the transformation.

The purpose of the reformat process is to create a common data form of the source data for the restructuring step in the conversion process. In the case of a simple sequential file, not under DBMS control, it may enter the conversion process at this point. Depending on the design of a system, this step may involve editing (i.e., encoding or recoding of items), limited data extraction, correction, and the like. Since the goal in this step is to create a data structure of the source data without the system-dependent information, the mapping between the input and the output of the reformat process can be considered to be generally one-to-one. While

![Figure 3—Schematic data conversion process](From the collection of the Computer History Museum (www.computerhistory.org))
this step looks simple functionally, its actual specification and implementation can be quite complex. For example, an application program may use the high order bits of a zoned decimal number for its own purposes, knowing that these bits are not used by the system. The specification of non-standard item encodings is a difficult problem in data conversion.

The load process is the counterpart of the unload process and needs no further clarification. Note that, however, the use of a common data form provides additional benefits, such as easing the portability problem.

The restructuring process is undoubtedly the most complex process of a generalized data conversion system. The languages for this mapping process can be quite different (for example, some procedural and other nonprocedural) and the models used to represent the data in the conversion system are also quite divergent. (For example, some use network structure; others use hierarchical structure.) More will be said on this topic later in this section.

Let us now turn to discuss the issue of implementation briefly. Generally there are two techniques. One can build the system using an interpretive approach or the generative approach. In the interpretive approach, the action of the system will be driven by the descriptions written in the system’s languages via the general interpreter implemented for the particular system. In the generative approach the data and mapping descriptions are fed into the compiler(s) which generates a set of customized programs executable on a certain machine. The merits of each of these approaches will also be discussed later in this section.

We shall next turn our attention to the tools that have been developed for data conversion. We shall first discuss the tools currently available and then the research and development work in progress.

Available conversion tools

Currently all available tools are limited in capability. Because it is impossible in this short report to provide an exhaustive survey of all the vendor developed conversion tools, we will highlight the spectrum of capabilities available to the user by providing examples from specific vendor shops.

The vendor’s repertoire of conversion tools begins at the character encoding level of data conversion with the provision of hardware/firmware options and continues through the software aids for conversion and restructuring of databases.

Depending on a diversity of conditions, the need to develop software tools varies from vendor to vendor. Probably the most prevalent file conversion tool is a COBOL foreign file processing aid. This type of facility allows the direct reading or writing of a particular class of files such as EBCDIC tapes or Honeywell Series 200/2000 files within COBOL. Although a relatively widespread facility, its capabilities are nevertheless limited. For example, some do not handle unlabeled tapes, while others cannot process mixed mode data types. Aside from the work of Bakkom and Behymer, which was aimed toward achieving a general conversion bridge with a particular vendor, to our knowledge there are no vendor supported generalized file translation tools.

In contrast to the above file translation tools, tools have been developed that have their main applications in a database environment. One example of a database conversion aid is the I-D-S/II migration aid provided by Honeywell. Because of the large volumes of data involved and the fact that the user cannot afford to shut down his whole data processing shop, a co-existence approach was adopted. The first step is to reformat the I-D-S/1 database into the I-D-S/II format, making the necessary data type conversions and pointer mechanism adaptations. This step allows the database to be processed in the I-D-S/II mode, but not optimally.

Additional steps in this migration include the generation of the additional I-D-S/II pointer fields (I-D-S/II requires “Prior & Header” chain pointers, which are allocated in step 1 but not filled in) and the restructuring of the I-D-S/II (coexistence) to the more sophisticated capabilities of I-D-S/II.

Some database restructuring tools specific to a particular DBMS have been developed by DBMS users. One example of this type of tool is REORG, a system developed at Bell Laboratories for reorganization of UNIVAC DMS-1100 databases. REORG provides capabilities for logical and physical reorganization of a database using a set of commands independent of DMS-1100 data management commands. A similar capability has been developed at the Allstate Insurance Company.

In addition to the above, there are also software companies and vendors who will do a customized conversion task on a contractual basis.

Data conversion prototypes and models

Over the past seven years a great deal of research on the conversion problem has been performed, with the results summarized in Figure 4. Projects were initiated at the University of Michigan, the University of Pennsylvania, IBM, SDC, and Bell Laboratories, as well as by a task group of the CODASYL Systems Committee. In many cases there was interaction and cross-fertilization between these groups and some consensus on appropriate architectures for data conversion was reached. The individual achievements of these groups is discussed below:

The CODASYL Stored-Data Description and Translation Task Group

In 1970 the CODASYL Systems Committee formed a task group (originally called the Stored Structure Description Language Task Group) to study the problem of data translation. The group presented their initial investigation of the area in the 1970 SIGMOD (then SIGFIDET) annual Workshop in Houston. In 1972 the group was reformulated as the Stored-Data Description and Translation Task Group and a general approach to the development of a detailed model for describing data at all levels of implementation was presented. The most recent work of the group specifies the data conversion model and presents an example language

From the collection of the Computer History Museum (www.computerhistory.org)
Figure 4—Summary of the development of data conversion models and prototypes
for describing and translating a wide class of logical and physical structures. The stored-data definition language allows data to be described at and distributed to the access path, encoding, and device levels.

The University of Michigan

The nonprocedural approach to stored-data definition set forth by Taylor and Sibley provided one of the major foundations for the development at the University of Michigan (see Figure 4) of data translators. In concert with Taylor's language, a model and design for a generalized translation was initiated by Fry, et al.

The translation model was tested in a prototype implementation of the Michigan Data Translator in 1972, and the results of the next implementation, Version I, were reported by Merten and Fry. In 1974, the work of the Data Translation Project of the University of Michigan focused on the database restructuring problem. Navathe and Fry investigated the hierarchical restructuring problem by developing several levels of abstractions, ranging from basic restructuring types to low level operations. Later, Navathe proposed a methodology to accomplish these operations using a relational normal form for the internal representation.

Version II of the Michigan Data Translator was designed to perform hierarchical restructuring transformations, but the project did not implement it. Instead, the research was directed into the complex problem of restructuring network type databases. To address this problem Deppe developed a dynamic data model— the Relational Interface model—which simultaneously allowed a relational and network view of the database. This model formed the basis of the Version IIA design and implementation of generalized restructuring capabilities. Another component necessary for the development of a restructurer was the formulation of a language in which to express the source to target data transformations. This language, termed Translation Definition Language (TDL), evolved through each translator version beginning with a source-to-target data item "equate list" in the Version I Translator to the network restructuring specifications of Version IIA. While the initial version of the TDL was quite simplistic, the current version, the Access Path Specification Language, provides powerful capabilities for transforming network databases.

The University of Pennsylvania

Concurrent with the work at the University of Michigan, Smith at the University of Pennsylvania (see Figure 4), also took a data description approach and developed a stored-data definition language (SDDL) for defining storage of data on secondary storage devices, and a translation description language (TDL) and three levels of database structures, the logical, storage, and physical, are described using the SDDL. In order to describe the source-to-target data mappings a first order calculus language was used. Following from this work, Ramirez implemented a language-driven "generative" translator which created PL/I programs to perform the conversion. One of the first reports on the utilization of generalization translation tools was provided by

IBM Research, San Jose

In 1973 another major data translation research endeavor was initiated at the IBM Research Laboratory in San Jose, California. Researchers in this project—initially Housel, Lum, and Shu, later joined by Ghosh and Taylor—adopted the general model as specified in Figure 1 but made several innovations. First, in the belief that programmers know well the structure of the data in a buffer being passed from a DBMS to the application program, the group concentrated its effort on designing a data description language appropriate for describing data at this stage. Second, regardless of the data model underlying any DBMS, the data structure at the time it appears in the buffer of an application program will be hierarchical. The general architecture, methodology, and languages reflecting these beliefs is reported in Lum et al.

In addition, the group in San Jose felt that, while it is desirable to have a file with homogeneous record types, it is a fact of life that many of today's data are still in COBOL files in which multiple record types frequently exist within the same file. As a result the group concentrated on designing a data description language which can describe not only hierarchical records (in which a relational structure is a special case) but also most of the commonly used sequential file structures. This language, DEFINE, is described by Housel et al.

The philosophy of restructuring hierarchies is further reflected in the development of the translation definition language CONVERT, as reported by Shu et al. This language, algebraic in structure, consists of a dozen operators, each of which restructures one or more hierarchical files into another file. The language possesses the capability of selecting records and record components, combining data from different files, built-in functions (e.g., SUM and COUNT), and the ability to create fields and vary selection on the basis of a record's content (a CASE statement).

A symmetric process occurs at the output end of the translation system. Sequential files are created to match the need of the target loading facility. The specification of this structure is again made in DEFINE.

A prototype implementation, originally called EXPRESS but renamed XPRS, is reported in Reference 28.

System Development Corporation

Another restructuring project reported by Shoshani was performed at The System Development Corporation in 1974-1975. In order to avoid the complexities of storage structure specification (i.e., indexes, pointer chains, inverted tables, and the like) they chose to use existing facilities of the systems involved. In particular they advocated the use of query and load (generate) facilities of database management systems. However, when such facilities do not exist, reformatters from the source (e.g., index sequential file) to a standard form and from the standard form to same output file had to be used. Given that databases can be
reformatted to and from a standard form, they concentrated on the problem of logical restructuring of hierarchical data bases in this form.

The language used in the above project for specifying the restructuring functions (called CDTL—Common Data Translation Language) was designed to be conceptually simple. For the most part it provides functions for specifying a mapping from a single field (or combination of fields) of the source to a single field of the target. For example, while a DIRECT would specify a one-to-one mapping of source items to target items, a REPEAT would specify the repetition of a source item for all instances of a lower level in the target hierarchy. In both cases only the source and target fields need to be mentioned as parameters. In addition there are more global operations, such as the INVERSION operator, which causes parent/dependent record relationships to be reversed. The system also supported extensive field restructuring operators, where individual field values could be manipulated according to prescribed language specifications. Since most of these operators are local, there is the possibility that they could be used in combinations that do not make sense globally. Therefore a further component of the system was built to perform "semantic analysis," which checks for possible inconsistencies before proceeding to generate the target database.

Bell Laboratories

The Bell Labs data translation system ADAPT (A Data Parsing and Transformation system), currently under development, is a generalized translation system driven by two high-level languages.31 One language is used to describe the physical and logical format and structure of the data and to provide various tests and computations to be used while parsing the source data and generating the target data. The second language is used to describe the transformations which are to be applied to the source data to produce the target data. Extensive validation criteria can be specified to apply to the source and target data.

Two processing paths are available within the ADAPT system: a file translation path and a database translation path (see Figure 3). A separate path for file translation is provided in response to real-world considerations. Many types of conversion do not require the capabilities and associated high overhead involved in using a database translation path.

Related work

Additional research effort is being devoted to the development and acceptance of a standard interchange form. An interchange form would increase the sharing of databases and provide a basis for development of generalized data translators. The Energy Research and Development Administration (ERDA) has been supporting the Interlaboratory Working Group for Data Exchange (IWGDE) in effort to develop a proposed data interchange form. The proposed interchange form29 has been used by several ERDA laboratories for transporting data between the laboratories. Additional work on development of interchange forms has been pursued by the Database Systems Research Group at the University of Michigan.33

Navathe34 has recently reported a technique for analyzing the logical and physical structure of databases with a view to facilitating the restructuring specification. Data relationships are divided into identifying and nonidentifying types in order to draw an explicit schema diagram. The physical implementation of the relationships in the schema diagram is represented by means of a schema realization diagram. These diagrammatic representations of the source and target databases could prove to be very useful to a restructuring user.

**Application program conversion**

So far we have concentrated on the data aspects of the conversion problem; it is necessary to deal as well with the problems of converting the application programs which operate on the databases. Program conversion, in general, may be motivated by many different circumstances, such as hardware migration, new processing requirements, or a decision to adopt a new programming language. Considerable effort has been devoted to special tools such as those to assist migration among different vendor's COBOL compilers, and general purpose "decompilers" to have been developed to translate assembly language programs to equivalent software in a high level language. While progress has been made developing special purpose tools for a limited program conversion situation, little progress has been made in obtaining a solution to the general problem of program conversion. With this fact in mind, this section focuses on the modifications to application programs that arise as a consequence of data restructuring/conversion.

**Problem statement**

There are three types of database changes which can affect application programs:

- alterations to the database physical structure, for example, the format and encoding of data, or the arrangement of items within records.
- changes to the database logical structure—either
  1. the deletion or addition of access paths to accommodate new performance requirements, or
  2. changes to the semantics of data, for example, modification of defined relationships between record types or the addition or deletion of items within records
- migration to a new DBMS, perhaps encompassing a data model and/or data manipulation language different from the one currently in use.

The actual impact of these database changes on application programs is a function of the amount of data independence provided by the Database Management Systems. Data independence and its relationship to the conversion problem are discussed elsewhere.50 We assume here that data independence is not complete and that therefore some degree of

From the collection of the Computer History Museum (www.computerhistory.org)
Program conversion is required in response to database schema changes. In fact, whereas most commercial database management systems provide application programs with insulation from a variety of modifications to the physical database, protection from logical changes—particularly at the semantic level—is minimal. Examples of semantic changes that are likely to have a profound effect on application programs include:

- Changes in relationships between record types, such as changing a one-to-many association to a many-to-many association or vice versa.
- Deletion or addition of data items, record types, or record relationships.
- Changing derivable information ("virtual items") to explicit information ("actual items") or vice versa.
- Changes in integrity, authorization or deletion rules.

There are various properties of database application programs that greatly complicate the conversion problem. For instance many database management systems do not require that the record types of interest (or possibly even the database of interest) be declared at compile time in the program; rather these names can be supplied at run time. Consequently at the compile time, incomplete information exists about what data the program acts on. Other troublesome problems occur when programs implicitly use characteristics of the data which have not been explicitly declared (e.g., a COBOL program executes a paragraph exactly ten times because the programmer knows that a certain repeating group only occurs ten times in each record instance). Complexity is introduced whenever a data manipulation language is intricately embedded in a host language such as COBOL. The interdependence between the semantics of the database accesses and the surrounding software greatly complicates the program analysis stage of conversion. Because of these considerations, substantial research has been devoted to alternatives to literal translation of programs. In particular some currently operational tools utilize source program emulation or source data emulation at run time to handle the problem of incomplete specification of semantics and yet still yield the effects of program conversion.

Current approaches

In this section, we discuss two main techniques currently employed in the industry. These techniques are commonly used but unfortunately not documented in the form of publications.

DML statement substitution

The DML statement substitution technique, which can be considered an emulation approach, preserves the semantics of the original code by intercepting individual DML statements calls at execution time, and substituting new DML statement calls which are correct for the new logical structure of the database. Two IBM software examples which provide this type of conversion methodology are (1) the ISAM compatibility interface within VSAM (this allows programs using ISAM calls to operate on VSAM database), and (2) the BOMP/DBOMP emulation interface to IMS. This program conversion approach becomes extremely complicated when the program operates on a complex database structure. Such a situation may require the conversion software to evaluate each DML operation against the source structure to determine status values (e.g., currency) in order to perform the equivalent DML operation on the new database. Necessary for the generalization of this approach is the development of emulation code for the following cases: maintain the run time descriptions and tables for both the original and new database organizations, interpret all original DML calls, and utilize old-new database access path mapping description (human input) and rules to dynamically determine what set of DML operations on the new database are equivalent to each specific operation on the source database.

Although this approach is straightforward in concept, it has several drawbacks. The drawbacks can be categorized as degraded efficiency and restrictiveness. Efficiency is degraded primarily because each source DML statement must be mapped into a target emulation program, which uses the new DBMS to achieve the same results. The increased overhead in program size and/or processing requirements can be significant.

The drawback of restrictiveness comes about because the emulation approach inhibits the utilization of the increased capabilities of the new DBMS and/or data structure through the modeling of the old methods. Additionally dependence upon the old program semantics limits the sets of permissible new data structures must support all of the semantics of the source program if the source program is to continue to execute in the same manner. It should be noted that the rules can be quite complex, even for the limited situation of which the data structure changes preserve semantic equivalence. Therefore, in some instances, just the limited task of determining if a change in data structure (given no change in the data model) will support a set of source programs will be an extensive task.

Bridge program

The second method in use today is sometimes referred to as the Bridge Program Technique. In this technique, the source application program's access requirements are supported by reconstructing from the target database that portion of the source database needed. Data reconstruction is done by means of "bridge programs." The source program is then allowed to operate upon this reconstructed portion of the source database to effect the same results that would occur if the source database were not modified. Of course, a reverse mapping is required to reflect update and each simulated source database segment must be prepared before it is needed by the application program.

This approach suffers from the same types of disadvantages inherent in the emulation approach. Efficiency problems for complex/extensive databases and programs performing extensive data accessing can make this method prohibitively expensive for practical utilization. This technique is generally found as a "specific software package"
developed at a computer installation rather than as a standard vendor supplied package.

Current research

Differing from the DML statement substitution and bridge program techniques, current research aims toward developing more generalized tools to automatically or semi-automatically modify or rewrite application programs. The drawbacks of the existing approaches described above can be avoided by rewriting the application programs which would take advantage of the new structure and semantics of a converted database and by using a general system to do the conversion rather than using ad hoc emulation packages and bridge programs.

Research on application program conversion is still in its infancy. Consequently, there are very few published papers on this subject. A handfull of works are described here in the order of the dates of publication. Mehl and Wang presented a method to intercept and interpret DL/I statements to account for some order transforms of hierarchical structures in the context of the IMS system. Algorithms involving command substitution rules for various structural changes have been derived to allow the correct execution of the old application programs. This approach works only for a limited number of order transformation of segments in a logical IMS database. Since it is basically an emulation approach, it has the drawbacks discussed in the previous section.

A paper by Su gives a general model of application program conversion as related to database changes resulting from a database transformation. An attempt was made to identify the tasks required for the automatic or semi-automatic conversion of application programs due to database changes. Two main points are stressed in the paper: (1) the need for extensive analysis of an application program including the analysis of program logic, data variable relations, program-subprogram structure, execution profile, etc., and (2) the use of database translation operators to determine what program transformations are required to account for the effects of these operators. The idea of the use of a common language to describe the operations of source queries and the data translation statements is also proposed.

An approach to the transformation of DBTG-like programs in response to a database restructuring was proposed by Schindler. The approach is based on the concept of code templates, which are predefined sequences of host language—DML statements (roughly analogous to assembly language macros). Application programs can be written as nested code templates. The code templates are devised so that each one corresponds to an operator in the relational algebra. An application program is then mapped into a relational expression, transformations are performed on the expression to accommodate the database restructuring, and a new program is generated by mapping the transformed expression back into code templates. This approach suggests that a level of logical data independence may be achieved through current programming technology.

The work by Su and Reynolds studied the problem of high-level sublanguage query conversion using the relational model with QUEL as the sublanguage, DEFINE as the data description language and CONVERT as the translation language. Algorithms for rewriting the source query were derived and hand simulated. In this study, query transformation is dictated by the data translation operators which have been applied to the source database. The purpose of this work was to study the effects of the CONVERT operators on high-level queries. Only restricted types of QUEL queries were considered. It is clear from this work that a general program conversion system should separate the data model and schema dependent factors from the data model and schema independent factors; and an abstract representation of program semantics and the semantics of data translation operators need to be sought so that data conversions at the logic level (especially the type which changes the database semantics) and the DBMS level can be attempted.

Two independent works carried out about the same time by Su and Liu and House use a more general approach to the application program conversion problem. The former work is based on the idea that the same data semantics (a conceptual model) can be modelled externally by various existing data models (relational, hierarchical and network) using different schemas. Application programs are mapped into an abstract representation which represents program semantics in terms of the primitive operations (called access patterns) that can be performed on data entities and associations. Transformation rules are then applied on the abstract representation based on the types of changes introduced by the data translation operators. The transformed representation is then mapped into another intermediate representation (called access path graphs) which is dictated by the external model and specific schema used for the target database. This representation is then modified by an optimization component and used for the generation of target programs. It is stressed in this work that the semantics of both the source and target database be made explicit to the conversion system and be used as a basis for application program analysis and transformation. The conversion methodology described is for program conversion to account for data conversion at the logical level as well as the DBMS level.

The work by House is an extension of the work on application migration undertaken at the IBM San Jose Laboratory. This work uses a common language for specifying the abstract representation of source programs as well as for specifying the data translation operations. The language is a subset of CONVERT plus some of CODD’s relational operators. The operators of the language are designed to have simple semantics and convenient algebraic properties to facilitate program transformation. They are designed to handle data manipulation in a general hierarchical structure called a “form” as well as relational tables. In this system, program transformation is dictated by the data mapping operations applied to the source database. It is assumed in the proposed model that the inverse of these data mapping operators exists, i.e., the source database can be reconstructed from the target database by applying some inverse operators on the target database. More precisely, it is assumed that $M'(T) = S$ where $S$ is the source database, $T$ is the target.
Program conversion via decompilation is a technique whereby a database application program is first transformed into an operationally equivalent higher order language or an abstract representation and then returned to a usable language level in a converted form. The transformation to a higher order language level is a decompilation process and the process of returning the program to a language level appropriate to conventional compilers is a compilation process. The underlying concept is that the decompilation to a higher order language can produce a functionally equivalent program that does not contain the DBMS, data model and data dependencies that inhibit the conversion process. That is, the decompiled program has the same "intent" while being unrelated to the changed DBMS environmental conditions. The changed environmental conditions should be easily incorporated into the program during the process of compiling the program back into a form appropriate to the new system.

There is some thought among researchers that this would be the preferred method to effect DML/host language program conversion. It should avoid many of the efficiency/restriction drawbacks inherent in current automated methods, while being more cost effective and less error prone than current manual methods (e.g., program rewrite).

One likely disadvantage to this method is that in order to use it to convert existing database application programs the programs may have to first be manually altered to place DML related code in a structured format. This disadvantage is to be expected because of the ambiguity inherent in the organization of DML/host language programs. However, the development of structured programming templates designed for DML related code should provide a means for creating programs that are convertible by the decompilation method. Structured templates might also provide the needed insight toward the development/selection of an appropriate high level language into which programs can be compiled. Some initial concepts of database program templates have been proposed by the University of Michigan.37

Current research directions

The current research has uncovered several problems which need to be further investigated before the implementation of a generalized conversion tool can be attempted. The following are the most important for future research:

Semantic description of database and application programs

Based on the work by Su and Liu45 and the study of the DPCTG group, it is quite clear that a program conversion system would need more information about the semantics of the source and target databases than the information provided by the schemas of the existing DBMS. Semantic information of the databases is an important source for determining the semantics of application programs which is the real bottleneck of the application program conversion problem. Future research needs to be conducted to (1) model and describe the semantics of application programs, (2) study the meaningful semantic changes to databases and their effect on application programs, and (3) derive transformation rules for program conversion which account for the meaningful changes. Several existing works on database semantics45-46 may provide a good basis for future works on this subject.

Equivalency of source and target programs

Data conversion may alter the semantic contents of the source database. A converted application program may or may not perform the identical operation on the target data as the source program on the source data. For example, it may not retrieve, delete, or update the same data as the source program because some records may be deleted and data relation may have been changed in data conversion. It is not clear at all how we can prove, in general, that a target program generated by a conversion system still preserves the original intent of the source program. Naturally, if the source data can be reconstructed from the target data without losing the original data relations and occurrences, we can establish the equivalence relation between the source and target programs based on the same effect they have on the source and target data.

Decompilation


From the collection of the Computer History Museum (www.computerhistory.org)

Conversion aids

A system which provides assistance to conversion analysts would seem to be a practical tool and a feasible task. Given the information about data changes and semantics of the data, a system can be built to analyze application programs to (1) identify and isolate program segments which are affected by the data changes, (2) detect inefficient code in the programs, (3) produce a program execution profile which gives an estimate of the computation time required at different parts of the program, and (4) detect, in some cases, the program code which depends on the programmer's assumption of data values, ordering of records, record or file size, etc. The data obtained in 1 together with some on-line editing and debugging aids would speed up the manual conversion process. The data obtained in 2 and 3 would be useful for producing more efficient target programs and the data obtained in 4 would help the conversion analyst to eliminate the 'implicit semantics' in programs which makes the program conversion task (manual or automatic) extremely difficult. To assist the conversion analysts in identifying ramifications of changes to programs a more complete cross-referencing than that usually produced by today's compilers can be extremely helpful. An example of such a product is the Data Correlation and Documentation system produced by PSI-TRAN Corporation. One technique, sometimes used during a conversion process that has been initiated by a database structure change, is to alter the names of effected database items in the DDL only and use errors generated by the compiler to locate program segments which need to be changed. A more complete cross-referencing system would be a much better tool, if it were available.

Optimization of target program

As the result of data conversion, multiple access paths to the same data may occur. This is because redundant data may be introduced or new access paths may be added in the course of data conversion. In this situation, a conversion system will have the choice of selecting a path to generate the target program. The efficiency of the program during execution time may depend on the selection of optimized access path during program conversion. Also, for reasons of achieving generality, some program conversion techniques proposed convert small segments of programs or the equivalent of DML statements separately. It is necessary to do a global optimization or simplification to improve the converted program. Techniques for program optimization related to program conversion need to be investigated.

Analysis of prototype conversion systems

This section analyzes the state of the art of generalized data conversion systems. It summarizes what has been shown to be technically feasible and points out what has been learned in the various prototypes. The prototypes have yielded encouraging results, but some weak points have also emerged. Later sections list some questions that remain to be answered and comment on additional features that will be necessary to enhance usability and analyze some implementation issues which can affect the cases where a generalized conversion system can be applied.

Where do we stand

The prototype systems described earlier have been used in a few conversions. While some of these tests were made on 'toy files', a few of the tests involved data volumes from which realistic performance estimates can be extrapolated. This section will summarize the major tests that were done with each of the prototypes.

The Penn Translator

The translator developed by Ramirez at the University of Pennsylvania processes single sequential files to produce single sequential target files. Facilities exist for redefining the structure of source file records, reformating and converting accordingly. Conversion of the file can be done selectively using user-defined selection criteria. Block size, record size, and character code set can be changed, and some useful data manipulation can be included.

The translator was used in several test runs on an IBM/370 Model 165. The DDL to generated PL/1 code expansion ratio was 1:4, so coding time was reduced.

A further test of the Penn Translator was conducted by Winters and Dickey. An experiment was conducted comparing a conventional conversion effort against the Penn Translator (slightly modified). Two source files stored on IBM 1301 disks under a system written for the IBM 7080 using the AUTOCODER language were converted to two target files suitable for loading into IMS/VS. Much of the data was modified from one internal coding scheme to another. The conversion required multiple source files to multiple target files.

The conventional conversion took several months versus five months for the generalized approach, a productivity improvement of roughly thirty percent. Time for adapting the translator, learning the DDL, and adapting to a new operating system is included in the five month figure. Without these, an estimate of three months was made for the conversion using the generalized approach.

The SDC translator

The translator described in References 29 and 30 was implemented during 1975-1976. The translator could handle single, hierarchical files from any of three local systems—TDMS, a hierarchical system which fully inverts files; DS/2, a system which partially inverts files; and ORBIT, a bibliographic system which maintains keys and abstracts. Databases were converted from TDMS to ORBIT, from TDMS to DS/2, and vice versa, and from sequential files to ORBIT. TDMS files were unloaded using an unload utility. Target databases were loaded by target system load utilities. The total effort for design and implementation was about three man-years. The system was implemented in assembly language on an IBM/370 Model 168, and occupied about 40 K-bytes, not including buffer space which could be varied. The largest file tested was on the order of five million char-
acters and the total conversion time was about 1 minute of CPU time per 2.5 megabytes of data.

The work was discontinued in 1976.

The Honeywell translator

The prototype file translator developed at Honeywell by Bakkom and Behymer\textsuperscript{1} performed file conversions (one file to one file) among files from IBM, Honeywell 6000, Honeywell 2000, Honeywell 8200 sequential and indexed sequential files. Data types of fields could be changed as well as field justification and alignment. New fields could be added to a record and fields could be permuted within a record. File record format (fixed, variable, blocked, etc.) could be changed and a compare utility was available for checking the consistency of files with different field organizations and encodings. Tests of up to 10,000 records were run. Performance of 15 milliseconds per record was typical (Honeywell Series 6000 Model 6080 computer). The prototype has been used in a conversion/benchmark environment but has not been offered commercially.

The Michigan data translator

Version 11B, Release 1.1 of The Michigan Translator was completed for the Defense Communications Agency in October 1977.\textsuperscript{48} It offers complete conversion/restructuring facilities for users of Honeywell sequential, ISP, or I-D-S/I files. Up to five source databases of any type may be merged, restructured or otherwise reorganized into as many as five target databases, all within a single translation. Database description is accomplished by minor extensions to existing I-D-S DDL statements. Restructuring specification is easily indicated via a high level language. Tests performed to date included a conversion of a 150,000 record I-D-S/I database with a total elapsed time of 24 hours (500 milliseconds per record). A given translation can be broken off at any point to permit efficient utilization of limited resources and also protect against system failures. The user is provided with the capability of monitoring translation progress in real time.

XPRS

Test cases with the XPRS system have focussed on functionally duplicating real conversions which had been done previously by conventional methods. Several cases have been programmed. In each case at least two input files were involved. Generally there was a requirement to select some instances from one file, match with instances in another file, eliminate some redundant or unwanted data, and build up a new hierarchical structure in the output. In several cases there was a need for conditional actions based on flags within the data. In all cases, the XPRS languages were found to be functionally adequate to replicate the conversion. A productivity gain of at least fifty percent in total analysis, coding, and debugging time was achieved. Test runs were conducted on several thousand records. Performance was deemed adequate in that XPRS can restructure data at least as fast as it can be delivered from direct access storage. No detailed performance comparisons were made comparing XPRS-generated programs with custom written programs.

Questions remaining to be answered

Given that several prototype data translation systems are operational in a laboratory environment, there is a little question concerning the technical feasibility of building generalized systems. The remaining questions pertain to the use of generalized systems in "real world" data conversions involving a wide variety of data structures, very large data volumes, and significant numbers of people. Three major questions to be resolved are:

1. Are the generalized systems functionally complete enough to be used in real conversions, and if not, what will it take to make them functionally complete?
2. Can the people involved in data conversions use the languages? What additional features are necessary to enhance usability?
3. Overall, what is the productivity gain available with the generalized approach?

Within the next year, prototype systems will be exercised on a variety of real-world problems in data translation, and concrete answers to these questions should be available. The systems being further tested for cost-effectiveness are the Michigan Data Translator, the IBM XPRS system, and the Bell Laboratories ADAPT system.

To date, preliminary results have been promising. A significant sample size on which to do analysis of productivity gain should be available at the end of the year of testing.

A number of factors must be taken into account in measuring the cost-effectiveness of the generalized data translator versus the conventional conversion approach. These factors include:

- ease of learning and using the higher level languages which drive the generalized translators;
- availability of functional capability to accomplish real world data conversion applications within the generalized translators;
- overall machine efficiency;
- correctness of results from the conversion;
- ability to respond in timely fashion to changes in conversion requirements (conversion program "maintenance");
- debugging costs;
- ability to provide "bridge back" of converted data to old applications;
- ability to provide verification of correctness of data conversion;
- capabilities for detection and control of data errors.

The languages used to drive generalized data translators are high-level and non-procedural; they provide a "user-friendly" interface to the translators. Since the languages are high-level, programs written in them have a better chance of being correct. Experience to date with DEFINE and CONVERT, the languages which drive XPRS, has
also be evaluated. Experience to date indicates that the languages are easy to learn and use. However, it would be wrong to think that these prototypes are mature software products or that they can be used in all conversions. This for the target database design to change or for new requirements during the conversion period. In an environment where a generalized translator is used regularly as a tool for conversion, costs associated with the debugging phase should be decreased through the use of common software modules. It is unusual in the conventional approach for common conversion modules to be developed. Thus each new conversion system requires debugging.

Usability

The usability of generalized data translation systems must also be evaluated. Experience to date indicates that the languages are easy to learn and use. However, it would be wrong to think that these prototypes are mature software products or that they can be used in all conversions. This section discusses some of the unanswered questions with respect to usability of the current data conversion systems.

One question concerns the level of users of the generalized languages. Current prototypes have been used by application specialists and/or members of a database support group. The systems have not yet been used by programmers, and the question remains whether programmers (as opposed to more senior application specialists and analysts) will be able to use the systems productively. There is no negative data on this point; the systems have not been used widely enough.

At present, all the systems require a user to describe explicitly the source data to be accessed by the read step using a special data description language. These data description languages are generally easy to learn and use; they resemble statements in the COBOL Data Division. However, the writing of the description is a manual process which can be tedious because a person may have to describe a file with hundreds of fields. Ideally, a data conversion system should be able to make use of an existing data description, such as those existing in a data dictionary or a system COBOL macro library. As evidenced by the Michigan Data Translator, it is reasonable to expect that such an interface will be available as data conversion systems evolve. It should be pointed out, however, that a data dictionary or COBOL macro library link may not necessarily solve the problem. Data in current systems is not always fully defined to be converted. This is especially true with non-database files. In these files, data definition often is embedded in the record structures of the programs, and a full definition depends on a knowledge of the procedural program logic. Even with existing databases, some fields and associations may not be fully defined within the system database description. Thus, the user can expect a certain amount of manual effort in developing data definitions. If existing documentation is incomplete, this can be a time consuming task, though it probably must be done regardless of whether a generalized package is used or not.

Another area where a user may have to expend effort is in the unload step of the data conversion process. The data description languages used to drive the read step have a limited ability to deal with data at a level close to the hardware (e.g., pointers, storage allocation bit maps, etc.). It is generally assumed that a system utility program can be used to unload source data and remove the more complex internal structures. Another alternative is to run the read step on top of an existing access method or database management system with the accessing software removing the more complex, machine dependent structures. These approaches are acceptable in a great many environments, including most COBOL environments, but there may be cases where neither approach will work. For example, a load/unload utility may not exist, or a file with embedded pointers which was accessed directly by an assembly language program might not be under the control of an access method. For these cases, the user is faced with complexity during the unload step. The complexity associated with accessing the data would appear to be a factor for either the conventional methods or for the generalized approach. However, in cases such as those above, some special purpose software may have to be developed. It should be noted that some research has examined the difficulty of extending data description
languages to deal directly with these more complex cases. 
The conclusion is that providing the data description lan-
guage with capabilities to deal with more complex data 
structures greatly complicates the implementation and has 
an adverse affect on usability. Thus, special purpose unload 
programs will continue to be required to deal with some 
files.

Analysis of architectures

This section discusses some of the different approaches 
that have been taken in implementing the prototype data 
conversion systems. The objective is to analyze some of the 
performance and usability issues raised by the prototypes.

Two approaches have been used in the prototypes—gen-
erative approach in the Penn Translator and XPRS, and an 
interpretive approach in the Michigan Data Translator. In 
the generative approach, a description of the input files, 
output files, and restructuring operations is fed to a program 
generator. From these descriptions, special purpose pro-
grams are generated to accomplish the described conver-
sion. In both the Penn Translator and XPRS, PL/I is the 
target language for the generator. The generated PL/I pro-
grams are then compiled and run. In the interpretive ap-
proach, tables are built from the data and/or restructuring 
description. These tables are then interpreted to carry out 
the data conversion.

In data conversion systems, as in other software, an 
implementation based on interpretation can be expected to run 
considerably more slowly than one based on generation and 
compilation. Initial experience with prototype data transla-
tors has shown that there is much repetitive work, strategies 
for which can be decided at program compilation/generation 
time. Also, there is a good deal of low level data handling, 
such as item type conversions. Thus, those implementations 
based largely on an interpretive approach run more slowly, 
and the ability to vary bindings at run time does not appear 
to be necessary. Interpretation was chosen in the prototypes 
for ease of implementation, and in the future it can be ex-
pected that a compilation-based approach or a mixture of 
compilation with interpretation will be the dominant imple-
mentation architecture. However, for medium scale data-
bases, the machine requirements of the interpretive data 
conversion prototypes are not unreasonable, and overall 
productivity gains are still possible.

Performance measurements with conversion systems 
based on the generative approach indicate that generalized 
systems can be quite competitive with customized programs. 
In one case, the program generated by the data conversion 
system ran slightly faster than a "customized" program 
which had been written to do the same job. However, this 
example could well be the exception and it would be naive 
not to expect this in general. The reason generalized packages 
can be competitive is that they often have internal algorithms 
which can plan access strategies to minimize I/O transfer 
and/or multiple passes over the source data. "Customized" 
conversion programs written in a conventional programming 
language often are not carefully optimized, since the expec-
tation is that the programs will be discarded when the con-
version is done.

A second architectural difference involves the use of an 
underlying DBMS or not. In both the Penn Translator and 
XPRS, the generated PL/I program, then executing, ac-
cesses sequential files, performs the restructuring, and 
writes sequential files. On the other hand, the Michigan Data 
Translator functions as an application program running on 

a network structured database management system. Thus 
the interpreter makes calls to the underlying DBMS to re-
trieve data during restructuring and puts restructured data 
into the new database.

The two approaches offer different tradeoffs. For exam-
ple, the Michigan Data Translator can make use of the ex-
isting extraction capabilities of a DBMS and perform partial 
translations easily. In addition, since it operates directly 
within the network data model, a user does not have to think 
of "unloading" data to a file model and then reloading it 
back; rather, the user describes a network to network re-
structuring much more directly.

On the other hand, when converting non-database data to 
a database, the use of an underlying DBMS as part of a data 
translation implies a second order data conversion problem— 
the non-database data must be converted into the DBMS of 
the data conversion system, which may or may not be dif-

cicult. It can be difficult, for example, when the data model 
of the data being converted differs significantly from the 
data model of the DBMS upon which the conversion system 
is based. Also, the use of an underlying DBMS may also 
require more on-line storage, whereas the file oriented con-
version systems can be made to run tape-to-tape. This can 
be important in very large database conversions.

In the future, one can expect that data conversion systems 
will offer a variety of interfaces to accommodate various 
kinds of conversion situations. For example, it is possible 
to interface the "file-oriented" conversion systems to run 
as application programs on top of existing database 
management systems. It is also possible to develop "reader 
programs" to load non-database data into conversion sys-
tems based on a DBMS. In addition, more automated inter-
faces to data dictionary packages can be expected in order 
to improve usability and obviate the need for multiple data 
definitions.

One possible performance problem with generalized con-
version systems lies in the unload phase. For reasons of 
usability, generalized conversion systems usually rely on an 
unload utility program to access the source data, thus iso-
lating the conversion package from highly system specific 
data. A potential problem with this approach is that the 
unload package may not make good use of existing access 
paths or may tend to access the source data in a fashion 
which assumes that the data has recently been reorganized 
(with respect to overflow areas, etc.). In cases where the 
data is badly disorganized, a customized unload program 
which accessed the data at a lower level might run consid-
erably faster, and for very large databases might be the only 
feasible way to unload the data. It is not clear how common 
this case is, and one can usually make the argument that the
Lessening the conversion effort

In order to identify guidelines for both reducing the need for conversion and for simplifying conversions which are required, it is necessary to consider the entire application software development cycle. This is because poor application design, poor logical database design, inadequate use of and inappropriate DBMS selection each could lead to an environment which may prematurely require an application upgrade or redesign. This redesign could, in many cases, require a major database conversion effort.

The set of guidelines specified below is not intended as a panacea. Instead, it is meant to make designers aware of strategies which make intelligent use of current technology. It is doubtful that all conversions could be avoided if a project adhered strictly to these proposed guidelines. However, adherence to the principles set forth by these guidelines could certainly reduce the probability of conversion, and more importantly, simplify the conversions that are required.

With respect to application design and implementation, the more the application is shielded from system software and hardware implementation details, the easier it becomes for a conversion to take place. For example, a good sequential access method hides the difference between tapes, disks, and drums from the application programs which use the access method.

The logical database design should be specified with a clear understanding of the information environment. A good logical database design reduces the need to restructure because it actually models the environment it is meant to serve. Introduction of data dependencies in the data structure should, if possible, be kept to a minimum. An analysis of the tradeoffs between system performance and likelihood of conversion should definitely be made.

Selecting the wrong or non-optimal database management system, given the application requirements, is also a key problem which can lead to unnecessary and large conversion efforts. The prospective user of a DBMS should, for example, carefully evaluate the data independence characteristics of a proposed DBMS.

The underlying principle of the guidelines which follow is that decisions can be made at the system design and implementation stages which are crucial to the stability of the applications.

- Application design guidelines

Requirements analysis

Many of the decisions affecting the long-term effectiveness of the application system (database design as well as application programs) are made during the requirements analysis stage of system development. Questions such as what are the functional requirements of the application, who the database is to serve, how the database is to be used, what are the possible future uses of the data, and what are the performance constraints of the application answered at this stage. It is essential that the designer understand the information environment as much as possible at the outset in order to lessen the probability that frequent conversions will be necessary.

Requirements analysis should focus on information needs and should minimize constraints being imposed by the physical environment. Imposing physical constraints at this junction is dangerous since it can distort the designer’s view of the true objectives of the application system. The influence of the physical environment should be considered secondarily, in order that the designer be fully aware of the resulting compromises to the logical requirements. This is not intended to imply that consideration of the physical environment is unimportant. Indeed, if the physical environment is ignored the effect could be development of a set of requirements that are impossible to meet within existing physical and cost constraints.

Program design guidelines

There are three underlying principles motivating this discussion of application program design. They are:

- design for maintainability
- design for the application
- data independence
Keeping sight of all of these during the design of the application program will lessen conversion effects by rendering the application as free as possible from physical considerations.

Designing for maintainability implies that the application should be written in a high-level language with a syntax that permits good program structure. Structured programming techniques such as top-down program design and implementation should be used throughout. The system should be modular with relatively small, functionally oriented programs. The programs should all be well commented and organized for readability. Design reviews and program walkthroughs also help to expose errors in the overall design and "holes" in the application logic at an early stage. It has been well documented that these steps help in making program modifications a much easier task.

One error which is often made in designing programs in a DBMS environment is to let the capabilities of the DBMS drive the design rather than the application. This design error can yield programs which are unnecessarily dependent upon the features of a specific DBMS. For example, in System 2000 one can use a tree to represent a many-to-many relationship instead of using the LINK feature. The parent/child dichotomy that results is an efficient but arbitrary contrivance that cannot easily be undone later on. The key principle here is to concentrate on what results are desired rather than on the implementation details of achieving these results. Simplicity and generalization of the design will provide a very high level of interface to the application programmer which will, in turn, minimize the total amount of software, provide the greatest degree of portability, maintainability, device independence, and data independence.

Of extreme importance in program design is the notion of data independence, i.e., insulating the application program from the way the data is physically stored.

Layered design

In the area of application design, the motivating factor for mitigating the effects of a conversion is to insulate logical operations from physical operations. One of the concepts applied to achieve this is layered design. That is, designing the application as a series of layers, each of which communicates with the system at a different level of abstraction. One can visualize this as an "onion," with hardware as its core and layers of successively more sophisticated software at the outer layers. The user interacts with the outermost skin of the onion, at the highest level of abstraction.

If application programs are written at the outermost layers of the onion, then these programs are smaller, easier to understand, and therefore easier to modify or convert than programs written at lower layers. For example, introduction of a new mainframe will require conversion of the software which references the particulars of the mainframe. However, since the layers are constructed so that physical machine and device independence is realized above some level, only the software below that level is subject to modification. To the extent that application programs stay at the outermost layers (i.e., above the critical layer) reduced conversion effects can be achieved.

We can thus summarize the goal of program design as follows:

- to provide the highest possible application interface to the program
- to maximize program independence from the characteristics of the mainframe, peripherals, and database organization
- to maximize portability of the application program through the use of high-level languages
- to maintain a clean program/data interface

Programming techniques

The previous sections of this chapter have focused on the design decisions which should be made to alleviate the conversion problem. However, regardless of how noble these goals are, poor implementation decisions can go a long way towards diminishing the returns of a good design. Equally important to intelligent design is a set of programming techniques and standards which prohibit programmers from introducing dependencies in code. For example, a "clever" programmer may introduce a word size dependency in a program by using right and left shifts to effect multiplication and division. Of course, there are no hard and fast safeguards against using tricky coding techniques; an effort must be made to make the programmer conscious of the consequences of this kind of coding. In particular, a programmer should not be allowed to jump across layers of the onion, such as to use an access method to directly read or write databases.

Database design

Perhaps the most costly mistake a designer can make is an error in the database design because it has a direct effect on the information that is derivable and the application programs that are created. Incorrect or unanticipated requirements can lead either to information deficient databases or overly complex and general design. An inadequate logical design has the potential for complex user interfaces or extremely long access times. A poor physical design can lead to high maintenance and performance costs. Unfortunately, database design is still an art at the present time. Two surveys report the results in the area to date. Novak and Fry survey the current logical database design methodology and Chen and Yao review database design in general. The work of Bubenko in the development of the CADIS system and the abstraction and generalization techniques of Smith and Smith show promise.

An accurate logical design can still be unnecessarily data dependent. Dependencies are inadvertently or deliberately introduced in the interest of improving system performance. In essence, "purity" is compromised to gain processing efficiencies. Since optimization is a worthwhile goal, insist-
From the collection of the Computer History Museum (www.computerhistory.org)

From the collection of the Computer History Museum (www.computerhistory.org)

Data- and Program-Related Conversion 903

ing on absolute purity may be unreasonable. However, the database designer should at least be aware of contrivances and therefore be in a position to evaluate the relative effects a design decision may have. Designers should become sensitive to their decisions by asking: "How will the data model be affected by a future change in performance requirements? Have I done a reasonable job in insulating applications from data structure elements that are motivated strictly by performance considerations?"

Some examples of induced data dependencies in logical database design which may impact upon conversion are:

- The use of owner-coupled sets in DBTG to implement performance-oriented index structures or orderings on records.
- Storing physical pointers (or database keys) in an information field of a record.
- Combining segment types (in DL/I) to reduce the amount of I/O required to traverse a database.

DBMS utilization and selection

Selection of a DBMS can have a major impact on conversion requirements. Of importance in evaluating a DBMS is to consider products exhibiting the highest level user interface.

A high level DBMS is characterized by both a powerful set of functions and a high degree of data independence from the point of view of the application. With respect to functions, that is, the DML, the distinction between "high level" and "low level" has traditionally centered on whether the DBMS provides user operations on sets of records (select, retrieve, update, or summarize all the records or tuples which satisfy some conditions) or whether one is restricted to record-at-a-time processing ("navigation"). The DBMS with the "high-level" set operation approach is significantly more desirable than the navigational record-by-record approach.

DBMS prospects should evaluate the data independence characteristics of a proposed product. Systems are preferred which support an "external schema" or "subschemata" feature which permits the record image in the application program (the user work area) to differ significantly from the database format. However, the subschema concept is only one aspect of data independence. In general, it is necessary to determine in what ways and to what extent the application interface is insulated from performance or internal format options. For instance, will programs have to be modified if:

- a decision is made to add or delete an index?
- the amount of space allocated to an item is increased or decreased?
- chains are replaced by pointer arrays?

Other conversion related questions about DBMS product include the following:

- Are there adequate performance and formatting alter-

natives? Are there too many (i.e., unproductive or incomprehensible) tuning options? Are there adequate performance measurement techniques and tools to guide the exercise of these choices?
- Does the system automatically convert a populated database when a new format option is selected?
- Aside from tuning, does the DBMS gracefully accommodate at least simple external changes such as adding or deleting a record or item type?
- Are there other useful high level facilities associated with or based on the DBMS, such as a report writer, query processor, data dictionary, transaction monitor, accounting system, payroll system, etc.?
- Is there a utility for translating the database into an "interchange form", i.e., a machine independent, serial stream of characters?
- Is the vendor committed to maintaining the product across new operating system and hardware releases/ upgrades? Conversely, is the vendor prepared to support the product in order of released versions of the operating system, so the user will not be forced to upgrade?
- What hardware environments are currently supported and what is the vendor's policy regarding conversion to another manufacturer's mainframe?
- What programming language interfaces are available? Can the same DBMS features be used if there is a migration, say, from COBOL to PL/1?
- How intelligent is the system's technique for organizing data on the media? Specifically, will performance deteriorate at an inordinate rate as updating proceeds?
- How often will reorganization (cleanup) be required?
- Is the vendor's policy regarding conversion to another DBMS adequate for the anticipated long term requirements of the enterprise? What is the risk of having to convert to a new DBMS?
- Likewise, are the performance characteristics and internal storage structure limitations adequate to meet the long term requirements (response times, database sizes) of the enterprise?
- Are there facilities to assist the user in converting data from a non-DBMS environment or from another DBMS? For instance, can a database be loaded from one or more user defined files?

Impact of future technologies and standards on conversion

In this section we discuss trends in computer hardware technologies, DBMS software directions, and standards development, and consider their impact on data and program conversion. Our intention is to make the reader aware of what to expect in terms of conversion problems rather than give a complete assessment of future technologies. Therefore, we discuss only technologies and standards that will impact conversion problems.

The first three parts discuss the areas of hardware, soft-
ware, and standards and their impact on conversion in some detail. The last part summarizes the major points of our assessment without going into detailed reasoning.

Hardware and architectural technologies

The cost and performance of processor logic and memory continue to improve at a fast rate. As a result, overhead costs are more acceptable, especially when such costs save people’s time and work, and provide user oriented functions that do not require a computer expert. In particular, one can now think about using generalized conversion tools not only when it is required as a result of hardware or software changes, but also as a result of a changing application that requires a new more efficient database organization. What could have been a prohibitive cost for a database conversion in the past, may not be a major factor in the future.

At the same time, the cost/performance improvement contributes to the proliferation of databases and therefore accentuates the need of generalized conversion tools. The most cost effective is the process of accessing and maintaining data, the more data is collected on computers. Improvements in hardware (as well as software) technologies create more need for data and program conversion. In addition, the emergence of new technologies, such as communication networks, add another level of sophistication to the way that data can be organized and used. Distributed databases, where multiple databases (or subsets of databases) may reside on different machines, require tools for the integration and the correlation of data. Invariably, data will need to move from system to system dynamically, possibly moving between different hardware/software systems. In this environment generalized tools for dynamic conversion will become a necessity.

In recent years two promising approaches to data management hardware technologies have been pursued. One is the specialized data management machine and the other is the backend data management machine. As will be explained next, both approaches can help simplify the conversion problem.

The specialized data management hardware is based on the idea of using some kind of an associative memory device, a device that can perform a parallel access to the data based on its content. Such a device eliminates the necessity for organizing the internal structure of a database using indexes, hash tables, pointer structures, etc., which are primarily used for fast access. As a result, the data can be essentially stored in its external logical form, and the data management system can use a high level language based on the logical data structure only. The conversion process is simplified since data is readily available in its logical organization. Referring to the terminology used in previous sections, the functions of unloading and loading of the database can be greatly simplified. Also, no restructuring will be required because of a change in database use, since the physical database organization can be to a large degree independent of its intended use. In addition, the program conversion problem is simplified as a result of the program interfacing to the DBMS using a high level logical language.

Similar benefits can be achieved if backend machines are used. A backend machine is a special processor dedicated to managing storage and database on behalf of a host computer. The primary motive for the backend machine is to off-load the data management function from the host to a specialized machine that can execute this function at much lower cost. From a conversion standpoint, the separation of data management functions from the host promotes the need for a high level logical interface that provides the advantages discussed above. Another advantage is that it is possible to migrate from one host machine to another without effecting the databases and their management, alleviating the need for data conversion if the same backend machine is used with the new host.

Another hardware technology development is mass storage devices, such as the video disks. These devices would make it cost effective to store very large databases, in the order of $10^{10}$ characters. The problem of converting large databases is compounded by cost considerations of processing this large amount of data. As a result it is likely that these databases will tend to stay in the same environment for longer periods of time. The use of specialized data management machines or dedicated backend machines in conjunction with these mass storage devices can help postpone the need for database conversion.

Finally, we should mention that there is a growing use of minicomputers in supporting data management functions. DBMSs are now available on many minicomputers, and more development is forthcoming. The proliferation of minicomputers which support databases will only increase the needs for generalized conversion tools.

Software development trends

Much of the work over the last years in the data management area have concentrated on techniques that clearly separate the logical structure of the database from its physical organization. This concept, called “data independence” was introduced to emphasize that users need not be exposed to the details of the physical organizations of the database, but only to its logical relationships. This led to the development of data access and manipulation languages that depend on the logical data model only. The effect of this trend is similar to that of using specialized data management machines and backend machines discussed previously; namely, the simplification of the unload and load functions since the interface to the DBMS is provided at the logical level only, and the simplification in program conversion for similar reasons.

At the user end of the spectrum, it seems reasonable to assume that the diversity of data models (network, relational, hierarchies and other views that may be developed in the future) will be required for many more decades. This is especially true since there are problem areas that seem to map more naturally into a certain model. Furthermore, it is often the case that users do not agree on the same model.
for a given problem area. Obviously this state of affairs only accentuates the need to generalize conversion tools that can restructure databases from one model to another. Even with the development of large scale associative memories, data structures will likely provide economic rationally for their contrived use. Another possibility is the use of a common underlying data model that can accommodate any of the user views. However, this approach will still require some type of a dynamic conversion process between the common view and each of the possible user views.

**Standards development**

There is much work and controversy in developing standards for DBMS. Standards that are oriented to determine the nature of the DBMS are hard to bring about even in a highly controlled environment because of previous investments in application software and database development, and because of disagreement. For example, there is still much controversy whether the network model proposed by the CODASYL committee is a proper one. It seems reasonable to assume that there will always be non-standard DBMSS. Further, even if such a standard can be adopted, different DBMS implementations will still exist, resulting in different physical databases for the same logical database. In addition, one can safely assume that restructuring because of application needs will still be necessary, and that changes in the standard itself may require conversion. A standard that is more likely to be accepted is one that effects only the way of interfacing to a DBMS. In particular, from a conversion standpoint, a standard interchange data form (SIDF) will be most useful. A SIDF is a format not unlike a load format for DBMSs. Any advanced DBMS has a load utility that requires sequential data stream in a pre-specified format. If a standard for this format can be agreed upon, and if all DBMSs can load and unload from and to this format, then the need for reformatting (as described earlier) is eliminated. The conversion process can be reduced to essentially restructuring only, given that unload and load are part of the DBMS function. A preliminary proposal for such a standard was developed by the ERDA Inter-Working Group on Data Exchange (IWGDE). However, it is only designed to accommodate hierarchical structures. Consideration is now given to the extension of the standard to accommodate more general structures (i.e., networks and relations). We believe that there are no technical barriers to the development of a SIDF, and that putting such a standard to use would alleviate a major part of the data conversion process.

**Summary**

The reasoning for the points summarized below is given in the previous parts of this section. We will only state here our assessment of the impact on conversion problems.

a. Hardware development will increase the need for generalized conversion tools (in particular, proliferation of minicomputers, computer networks, and mass storage devices).

b. The reduction in hardware costs will make conversion costs more acceptable.

c. Special Hardware DBMS machine will simplify the conversion process (in particular, for load, unload functions, and program conversion) because they promote interfacing at the logical level.

d. Software advances will not eliminate the need for conversion but can simplify the conversion process in a similar way as C.

e. Multiplicity of logical models are likely to exist, thus adding to the need of conversion tools between models.

f. Standards will not eliminate the conversion problem. Even if a standard is followed, the implementations would be different. Also, it is likely that non-standard DBMS will always exist.

g. Standards can greatly simplify conversion. In particular, a standard for interchange data form is likely to come about (will simplify load and unload and eliminate reformatting).

What can we expect in the next five years and beyond in the database conversion area? The state-of-the-art has advanced enough to date to give hope for generalized tools. Within the next five years we can expect more generalized conversion systems to become operational, but some additional work will be required for moving them from one environment to another. We can expect to have a standard form developed and agreed upon. It will probably take longer before manufacturers will see the benefit of adopting a standard form and provide load and unload facilities using it. However, we can expect them to provide some conversion tools to convert databases from other systems to their own. It will probably take as much as ten years before a generalized converter is available commercially, with manufacturers adhering to a standard form. Another area of concern is the application program conversion that is required as a result of database conversion. This topic was discussed in detail earlier. This is a difficult technical problem even within one data model, and still requires much research. It is hard to expect that a generalized solution for this problem will be achieved within the next five years. However, this problem will be elevated to a large extent as hardware and software development trends promote interfacing at the logical level.

**BIBLIOGRAPHY**


32. ERDA Interlaboratory Working Group for Data Exchange (IWGDE), Annual Report for Fiscal Year 1976, NTIS LBL-5329.


