A high-level data-base system, the Peterlee Relational Test Vehicle (PRTV), provides flexible, interactive data-base support and functional extensibility. The user sees the system primarily through a programming language called ISBL, which is designed for manipulating bulk data held in relations. PRTV is not a full-fledged data-base system, but rather an evolving prototype which is expected to aid in solving some of the problems that have been encountered in using relational data bases. PRTV embodies research both in data-base language design and in efficient implementation techniques.

The Peterlee Relational Test Vehicle—a system overview

by S. J. P. Todd

A high-level data-base system is one in which applications are expressed in a language meaningful to the user, and in which data are represented and manipulated in a natural way. At IBM's United Kingdom Scientific Centre in Peterlee, County Durham, a major research aim is to understand the problems associated with designing and using such systems for general applications. A number of software prototypes have been implemented to test various designs. One of those prototypes, the Peterlee Relational Test Vehicle (PRTV), is the subject of this report, which covers both internal and external features of particular interest in data-base systems design.

PRTV is not a full-fledged data-base system; many features essential to such a system, such as backup and recovery, have not been implemented. This report is intended to address problems of greater significance than PRTV itself, however, and it is believed that continued work with the evolving prototype will provide answers to some of those problems. Indeed, it is likely that one outcome of the continued use of PRTV will be that more problems will become apparent.

major aspects of PRTV

PRTV is an interactive data-base system intended to be used either as a stand-alone system for simple data bases or as a data subsystem for an applications system. The main objectives are high-level, flexible data-base support and functional extensibility. High-level support is provided by concepts based on the relational model. To give maximum flexibility in using the data

base, relations are treated as named variables by the user language, ISBL (Information System Base Language). New relations can be created and assigned at will. Extensibility is provided by allowing user written PL/I functions to be added to the system. PRTV also has a flexible method for defining multiple views of data.

PRTV is designed for efficiency in that instead of each user relation having its own file, there is flexibility in mapping between system files and user relations. This design allows greater freedom in the storing of data, and it allows the system to implement user requests more efficiently by reordering commands and choosing the most suitable access paths.

The system is modular, so small pieces can be changed in order to test new ideas. In addition, all routines that deal with the world outside the PRTV system—the operating system, terminals, card readers, printers, and disk I/O—have been implemented as small utility routines that can be converted readily for different operating systems and devices.

Use of the relational model as a formal background to high-level data bases was proposed by Codd,[1] who also discussed the relational algebra. Among the first implementations was the IS/1 system at Peterlee.[2] PRTV is based on experience with that prototype.

Concepts and facilities

The user sees the PRTV system principally through the Information System Base Language (ISBL), which is designed for manipulating bulk data held in relations. It provides for variables, expressions, and assignments in much the same way as do conventional programming languages such as PL/I. All variables denote relations, and the only operations that can be used in expressions are those that produce relational results.

In addition to assignment statements, ISBL provides control statements for creating new domains, sharing relations among users, and similar functions. For cases that cannot be handled by a relational algebra operating on stored relations, ISBL permits escape to previously linked PL/I extensions. There are standard extensions for the entering and listing of relations and for the basic arithmetic and string operations. A user can add other extensions as required. These extensions can themselves issue ISBL statements; they can also access data from a relation.

ISBL does not have flow or control statements such as DO WHILE or GO TO.

Figure 1 Relations can be thought of as tables. This example represents the information that book number 5 is Austen's *Persuasion*, and number 7 is Goethe's *Faust*.

ACQ_NO	AUTHOR	TITLE	← SELECTORS
ACQ_NO	NAME	NAME	← DOMAIN NAMES
5	Austen	Persuasion	
7	Goethe	Faust	

structures

The basic units understood by PRTV are called *objects*. Objects are grouped into sets called *domains*, each of which has a name. For example, in a library application there might be a domain called NAME, consisting of the titles of books and the names of authors and borrowers, and there might be another domain, ACQ_NO, consisting of acquisition numbers.

Every object is held in the system as either numeric or character data and is said to have *data type* N or C. All objects in a domain must have the same data type, which then is the data type of the domain. A new domain can be created at any time. For example, the ISBL statement

```
CREATE DOMAIN NAME, C
```

would be used to create the domain called NAME with data type C (character objects).

Information connecting the objects is held in *relations*. A relation is like a table, as shown in Figure 1. This relation represents the information that book number 5 is Austen's *Persuasion*, and number 7 is Goethe's *Faust*. Each row of the table is a *tuple* of the relation, and each column is a *component*. All elements in a column must be drawn from the same domain. The list of components from which the objects of a tuple or relation are drawn is the *relation type* (or simply *type*) of the tuple or relation.

A relation can be assigned to a named variable, but intermediate values in expressions can be left unnamed. The relation discussed above might be called BOOKS.

Each component of a relation is identified by a name called a *selector*. Often the selector is the domain name, but when the same domain underlies several components, the selectors must differ from the domain name. Usually the domain names are not included in the tabular representation (see Figure 2).

Figure 2 In a relation, the order of tuples and components is arbitrary. The table shown below represents the same relation as the table in Figure 1. The domain names are omitted; only the selectors are shown.

TITLE	AUTHOR	ACQ_NO
Faust	Goethe	7
Persuasion	Austen	5

A relation is a set of tuples, and the order of the tuples is arbitrary, as is the order of the components. Thus the table in Figure 2 represents the same relation as the table in Figure 1.

To describe a relation, the selector and domain name must be given for each component. Each domain name/selector pair is separated by a comma, and the domain name is separated from the selector by a colon. When the names are the same, only one need be given. Thus the relation in Figure 1 would be described by

```
BOOKS(ACQ_NO,NAME:AUTHOR,NAME:TITLE).
```

Tuples cannot be repeated in a relation, so in a relation of the type described above, there is no way to represent two copies of, say, *Persuasion*, both with the same acquisition number.

The number of components in a relation or tuple is the *degree*, and the number of tuples in a relation is the *cardinality*.

operations

There are six main operations in ISBL that act on relations to produce other relations. The operations are selection (designated by :), projection (%), union (+), intersection (.), difference (−), and join (*). These are the operations traditionally associated with a relational algebra, but their action has been generalized.[3] This simplifies both the expression of programs in the language and their implementation. The operations that give degree, selector, domain, and cardinality information about relations also return relations. This feature simplifies the language. The main operations are described more fully below.

Selection (:) acts on one relation and creates a new relation of the same type. The new relation is a subset of the old, all the tuples satisfying some criterion called a *filter*. Filters can contain comparisons between the objects in two components of a tuple, or between an object from the tuple and a constant. The objects being compared must be from the same domain. If the domain is numeric, the comparisons can be $=$, $>$, $\geq$, $\neg =$, $<$, or $\leq$. For character data, only $=$ and $\neg =$ are allowed. Filters can also contain many comparisons arbitrarily tied together by ε (*and*), $|$ (*or*), $\neg$ (*not*), or (). For example:

```
BOOKS : ACQ_NO = 5
```

selects the tuples for books with an acquisition number of 5, and

```
BOOKS : (ACQ_NO = 5 | ACQ_NO = 7) ε AUTHOR = 'Austen'
```

selects tuples for books by Austen with acquisition number either 5 or 7.

Projection (%) also acts on one relation to produce another. For each tuple in the original, the result contains a tuple with the components renamed or only some components present. A *projection list* specifies the selection of components and their new names. It contains selectors from the input relation, each optionally qualified by a new selector for the corresponding component in the resulting relation. To rename some components and leave the remainder as before, a list is given of the selectors to be changed with their new names, followed by, . . . (meaning *and so on*). For example:

```
BOOKS % AUTHOR,TITLE
BOOKS % AUTHOR − >WRITER,TITLE
BOOKS % AUTHOR − >WRITER, . . .
```

The first expression creates a relation of degree 2 with selectors AUTHOR and TITLE. The relation denoted by the second expression is similar except that the AUTHOR component is renamed WRITER. The third expression yields a relation similar to books, but with the AUTHOR component renamed as in the second.

Because two distinct tuples of the input relation may become identical on projection, the cardinality of the result of a projection may be smaller than the cardinality of its input. This effect, a natural result of the set-theoretic nature of relations, is akin to the purging of files.

The operations *union* (+), *intersection* (.), and *difference* (−) depend on the relation as a set of tuples. Each operates on two relations to produce a third. The result of a union is a relation that contains all the tuples appearing in either operand. An intersection produces a relation containing only tuples that appear in *both* operands. In either operation, the input relations must be of the same type, which becomes the relation type of the result. For example, the statement

```
BOOKS + NEW_BOOKS
```

produces the set of tuples for all books now in the library, and

```
BOOKS . NEW_BOOKS
```

produces a set containing only tuples from the relations BOOKS and NEW_BOOKS.

The operation difference finds tuples in the first relation for which there is no tuple in the second relation that matches in the components with matching selectors. The result is called a *difference on the common selectors*. (When all the components match for the relations, this operation becomes the conventional relational difference.) For example, a library has the relation

```
LOANS(ACQ_NO, NAME:BORROWER, DATE:DATE_OUT).
```

Then the statement

```
BOOKS − LOANS
```

produces a difference on ACQ_NO. It gives the acquisition number, author, and title of all books not currently on loan.

The final operation, *join* (*), also produces a new relation from two operand relations. In the most extreme case, every tuple in the first relation is paired with every tuple in the second. For each pair, a new tuple is created with all the objects from both tuples. This is called a *concatenation* of the contributing tuples. All these new tuples together form the result of the join. The cardinality of the result is the product of the cardinalities of the input relations, and the degree of the result is the sum of the degrees of the inputs. This form of join is a *full quadratic join.*

More common is the *natural join*, or *equijoin*. If selectors from the two relations match, tuples are put into the result only if the values for those selectors in the contributing tuples also match. The concatenated tuple holds only one occurrence of the selector, which contains the common value. The result is a *join on the common selectors.* Its cardinality can be any value from zero to the product of the input cardinalities. The degree of the result is the sum of the input degrees less the number of matching selectors. When the two relation types are the same, an equijoin degenerates into an intersection. In the library example, the statement

```
BOOKS * LOANS
```

produces a join on ACQ_NO, combining information from both relations about the books on loan. The result is a relation of degree 5 with selectors ACQ_NO, AUTHOR, TITLE, BORROWER, and DATE_OUT.

The automatic matching of selectors in the join and difference operations sometimes associates selectors that are required to be different, or fails to associate those that should be associated. The *rename* option of the projection operation is used to overcome this problem. To find pairs of books by the same author, for example, the author/title part of BOOKS is joined to itself on author. To avoid joining on title, the title components are renamed TITLE1 and TITLE2. This operation gives triples of author with two titles, from which the tuples with identical titles are eliminated. In ISBL, the operation appears as follows:

```
(BOOKS%AUTHOR,TITLE − > TITLE1) *
    (BOOKS%AUTHOR,TITLE − > TITLE2) : TITLE1 ¬=TITLE2.
```

user extensions

Relational operations provide a convenient method of manipulating and coordinating data, but they are incapable of carrying out computations. Also, a data base cannot provide listing and

data-entry services precisely tailored to user requirements. Rather than incorporating these services into ISBL, PRTV provides mechanisms for escape to two types of user extensions. One acts on a single tuple at a time; the other allows actions across sets of tuples. Tuple-at-a-time extensions are simpler to write, and they operate more efficiently, but they are not so powerful as general extensions.

Procedures are provided in PRTV to simplify the linking of new user extensions into the system. For a tuple-at-a-time extension, the user enters the names and data types of the parameters and the body of the PL/I code. The PL/I code is completed automatically by the addition of the procedure and declare statements for the parameters. Then it is compiled and linked into the system, and the directory of functions is updated.

tuple-at-a-time extensions

Tuple-at-a-time extensions are used for two kinds of operation: providing computed fields and providing user defined selection criteria. The user writes a PL/I program that accepts as input only the appropriate elements from a single tuple. It returns either the computed elements or a flag (the *predicate flag*) indicating whether the tuple satisfies the criteria. PRTV chooses the elements to be passed to the function from the tuple according to the ISBL statement used to invoke it, and it provides the control to call the function for each tuple of a relation. An example is the PL/I procedure ISIN:

```
ISIN: procedure (a,b) returns (bit);
      declare (a,b) character (*);
      declare predicate_flag bit;
      predicate_flag = (index (b,a) ¬=0);
      return (predicate_flag);
      end;
```

The last three lines are the only ones entered by the user. This procedure returns true only if character string *a* occurs in string *b*. Its use in ISBL is illustrated by the statement

```
LIST BOOKS * ISIN('Relation',TITLE)
```

which selects books with the character string *Relation* in their titles.

```
LIST BOOKS * ISIN(AUTHOR,TITLE)
```

is an expression that might be used by egocentric authors.

A computed field might be provided by a function DUE, which, given the loan date of a book, would return the due date. Then the statement

```
LOANS * DUE(DATE_OUT | DATE:DATE_DUE)
```

produces a relation with three components: ACQ_NO, DATE_OUT, and DATE_DUE. (The input and output parameters are separated by |, and for the output, both the domain (DATE) and selector (DATE_DUE) must be given.) The join notation is important, since ISIN is a procedure whereas ISIN(AUTHOR,TITLE) is a relation with selectors AUTHOR and TITLE (Hall et al.[3]).

general extensions

General extensions call the ISBL interpreter recursively, or they use the relational file interface to access and write data-base data one object at a time. They are used for interaction with the outside world (listing, data entry), for computations across tuples (subtotal), and for macro-type operations (definition of division in terms of other relational operations). Character strings are passed into the PL/I program from the ISBL interpreter. These strings can be treated in any way—for example, as a relation name or as a list of selectors.

The standard system utility ENTER accepts as parameters the name of a relation to be entered and the components it is to be given. The ISBL call to enter the BOOKS relation is

```
CALL ENTER(BOOKS|ACQ_NO,NAME:AUTHOR,NAME:TITLE).
```

Another extension is SUBCOUNT, which counts the occurrences of tuples for each value in a set of components. For example, the statement

```
CALL SUBCOUNT(LOANS|BORROWER|RESULT)
```

assigns to RESULT a relation giving the number of books that each borrower currently has on loan.

relational files

Relational files are used to move data between the data base and a program. There are two kinds of relational file:

- A *relational read file* is a "snapshot" of the data in a relation. The tuples are ordered and the file has a cursor. A relational expression can be turned into a file, in which case the tuples are transferred one at a time from the file to the program.
- A *relational write file* is used for transforming data from PL/I into the data base. The file is written tuple by tuple, and when the file is closed, it is given a name and turned into a relation.

It is important to note that relations, relational read files, and relational write files are all distinct. A tuple cannot be read from a relation, for example, nor can a relational write file take part in a union.

General extensions can also create and manipulate relations using ISBL. An ISBL statement is built up in the program, then

submitted to the system recursively using the PL/I statement CALL XEQ. General extensions that are included as standard in PRTV are LIST, ENTER, SUBTOTAL, and DIVIDE.

other features of PRTV

Other points of especial interest in the use of PRTV are discussed below:

- Update. ISBL does not have an update capability in the normally accepted sense, but the value of a relation variable can be changed by an assignment into it. For example, the statement A = A + S would insert a set (S) of tuples into a relation (A).

  An update that is equivalent to changing a field of a record can be accomplished only by using complicated expressions, or by PL/I extensions using relational files. An update capability eventually should be modeled at the ISBL level. At present, the implications beyond simple CHANGE or INSERT commands are not fully understood, so no update facility has been implemented.

- Workspace. When a user signs on, PRTV gives him an empty workspace. The workspace is a temporary (one-session) association of relation identifiers with user names. The names can be loaded explicitly from the user's workspace index, but they will be loaded implicitly if a referenced variable is not in the workspace. Unless the user explicitly requests that a change be made permanent, the result of any update or assignment will be limited to the workspace. Thus the workspace provides a convenient means of testing possible changes without altering the operational data. Workspaces cannot be saved from one session to another, as they can in APL.

- Multiple users. ISBL enables different users to work with the same data, each having access to different parts of the data. When a user signs on, he gives his name and a password to identfy a set of named relations belonging to that user. New relations created are private, but they can be shared explicitly with other users. Concurrent use of a single data base by many users is not supported.

- Variable binding. When a variable is used in an expression, ISBL allows binding both by value and by name.

  Binding by value is the default binding: the current value of the named relation is found and inserted into the expression. If the expression is used in an assignment, any subsequent change of value in the original relation is not reflected as

a change of value in the assigned relation. The effect is the same as the use of a variable on the right-hand side of an assignment in PL/I.

If an expression specifies *binding by name*, the named relation is not evaluated at the time. Instead, its name is held in a procedure which represents the expression. If the expression is used in an assignment, any subsequent change of value in the relation bound by name is reflected as a change of value in the assigned relation. The relation bound by name is thus evaluated whenever the assigned relation is used. Binding by name can provide different (read-only) views of the same data. For example, with the relations BOOKS and LOANS, the expression

```
FULL_LOANS = N!BOOKS * N!LOANS
```

causes changes in BOOKS or LOANS to be reflected automatically in FULL_LOANS. Relations such as FULL_LOANS, whose values depend on the values of other relations, are called *defined relations*.

- Excluded features. PRTV does not attempt to provide relational calculus or interfaces for inexperienced users, nor complex data-entry or report-generation features, nor nonrelational views of data other than relational files. All of these features can be provided by systems using PRTV as a subsystem. Integrity and backup features are also excluded.

examples of PRTV use

The following example shows how relations might be used in day-to-day processing in a simplified library system. The library might keep two permanent relations:

```
BOOKS(ACQ_NO,NAME:AUTHOR,NAME:TITLE)
LOANS(ACQ_NO,NAME:BORROWER,DATE:DATE_OUT).
```

BOOKS would list all books owned by the library, with acquisition number (assigned by the librarian), author, and title. LOANS would give the names of borrowers of books with a particular acquisition number, as well as the dates on which the books were taken out. Every day, three relations would be collected by the data entry routine:

```
LOANS_TODAY(ACQ_NO,NAME:BORROWER)
RETURNS_TODAY(ACQ_NO)
NEW_BOOKS(ACQ_NO,NAME:AUTHOR,NAME:TITLE).
```

The end-of-day processing would be as follows:

```
1 LOGON  LIB,LIB
2 BOOKS = BOOKS + NEW_BOOKS
3 LOANS_TODAY1 = LOANS_TODAY * <DATE=741121>
4 LOANS = LOANS + LOANS_TODAY1 - RETURNS_TODAY
```

```
5 PUT BOOKS
6 PUT LOANS
7 LOGOFF
```

The librarian signs on in step 1. Step 2 is the updating of total acquisitions by a union of books on hand and newly acquired books. Step 3 adds the date (in this case, 21 November 1974) to today's loans, to provide complete information for the permanent file. In step 4 the LOANS relation is updated by making appropriate insertions and deletions. (This step does not allow for books returned and taken out again the same day.) Since the updates have been made in the workspace, steps 5 and 6 are needed to enter them into the data base. When the librarian signs off in step 7, temporary relations such as LOANS_TODAY! are destroyed.

Not all details, such as the writing of user functions, are mentioned here, but the example indicates the advantages of PRTV for quickly writing new data-base applications. The advantages are even greater for nonstandard queries. Most queries can be phrased in one or two statements. For example:

Query Who has taken out *Persuasion*?

```
ISBL   LIST LOANS * BOOKS : TITLE='Persuasion'
```

Query What is the book by Austen about something-or-other Abbey?

```
ISBL   LIST BOOKS : AUTHOR='Austen' * ISIN('Abbey', TITLE)
```

Query What books by Austen are in at present?

```
ISBL   IN = BOOKS − OUT
       LIST IN : AUTHOR='Austen'
```

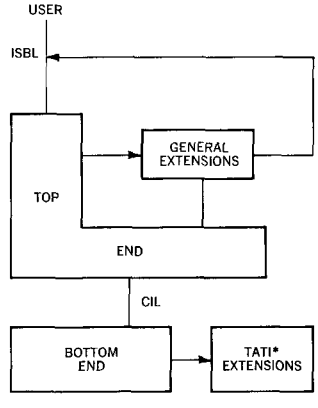
This example goes through an intermediate step. The relation IN holds information about all the books that belong to the library but are not currently on loan.

The example also illustrates the value of providing multiple views using the bind-by-name facility. If questions of the type "What books by Austen are in at present?" are anticipated, a definition of the relation IN can be prepared in advance, as

```
IN=N!BOOKS−N!OUT.
```

Because IN is only defined using bind-by-name, it is not evaluated until required. Thus extra disk space is not needed, and up-to-date values are obtained.

Figure 3 The implementation of PRTV.



Implementation

PRTV is implemented in two major sections called the *top end* and the *bottom end* (see Figure 3.) The top end is like an in-

terpretive compiler. It deals with syntax analysis, naming of relations, semantic checking of relation operations, and mechanisms for escape to general user extensions. It also deals with the presentation of relational files to the user program. The bottom end is a suite of subroutines that deal with the handling of records representing tuples. It is the bottom end that carries out the hard work of operations such as union and join and the storing of large quantities of data. The bottom end also handles the calling of tuple-at-a-time extensions. The interface between the top and bottom ends is the Common Intermediate Language (CIL).

the CIL interface

The CIL facilities form a record- and file-oriented access method designed to make the implementation of an ISBL-type language easy. The CIL user recognizes two basic data-set types: the physical data set, or *brick*, and the logical data set, or *stream*. Both appear as homogeneous sequential files. (Throughout this section, *user* means the user of CIL. In the full PRTV system, this user is the top end.)

All writing at the CIL level is done directly into a brick, and all reading is done from a stream. The simplest form of stream allows direct reading from a brick; but in general a stream is more complex, allowing for reading from a union of two bricks, for example.

One of the first things a CIL user does is write a set of records, which are stored in a brick. When the entire set has been written, the bottom end (on a CLOSE command) returns a numeric identifier to the set—22, for example. The user can write as many bricks as required, and for each, a unique numeric identifier is assigned.

Streams are used to read back data stored in the data base. If the user wishes to read back the data in a brick, a stream is opened giving the brick identifier, and the user goes through the records sequentially. The system also provides more complex streams created by combining several bricks. For example, the user writes a brick with identifier 45, as well as brick 22; all the records from both bricks are to be read back. A stream is opened giving the string +D22D45, which specifies the required set of records, and the stream is read sequentially as before. The string that specifies a stream is a *cilstring*.

A cilstring is a prefix Polish expression involving bricks and streams. Seven basic operations are used to combine streams to form new streams. Six correspond to the relational operations: union, intersection, difference, selection, join, and reorder. These operations work on ordered streams, not sets. There is an important difference, which is most significant in the operation

projection. A low-level projection can cause a need for a sort. Therefore the low-level projection is given a different name, *reorder*.

The seventh stream function (F) implements tuple-at-a-time extensions; it reads through an input stream, and for each record it passes the appropriate fields to the user function. It then either constructs a new record with the returned computed fields included, or decides, according to the predicate flag, whether or not to pass the record to the output stream.

There is one special function, read (D), which converts a brick to a stream by reading through the records.

All the above operations can be combined in an arbitrarily complex manner to construct a stream from a set of bricks.

Following are some examples of cilstrings:

D22	brick 22
+D22D45	union of bricks 22 and 45
%D45C2E	take second column only from brick 45
;+D22D45=C217E	select from union of 22 and 45, where column 2 equals 7

In all cases, a brick is read using the read operation (D). If a stream for a simple brick is required, the cilstring consists simply of the brick identifier prefixed by D.

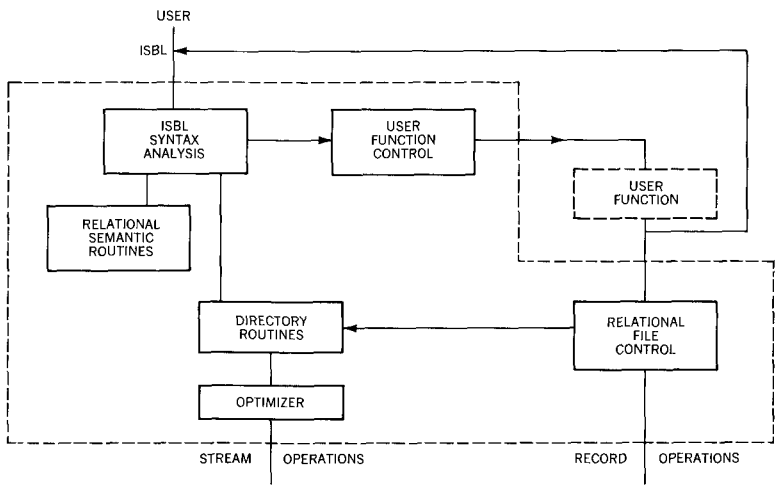
The cost of creating a stream for a complex cilstring may be great, so such a stream should not be recreated if the stream is to be used several times. Therefore a command is provided in PRTV to store the records of a stream in a brick. The command accepts a cilstring and returns a brick identifier. Another command is provided for the deletion of bricks after use.

Character-string data are not held directly in bricks, but in separate disk areas called *value sets*. Bricks contain pointers to the value sets. For efficiency, distinct sets of character data are held in different value sets. Commands are available to create and destroy value sets, and the OPEN command for writing a brick has a parameter that specifies which value sets are to be used. Numeric data are not held in value sets.

the top end

The ISBL interpreter consists of six main components. As shown in Figure 4, they are the ISBL syntax analyzer, directory routines, user function control, relational semantic routines, relational file control, and optimizer. The top end is implemented mostly in the string processing language MP/3,[4] but some of it is written in PL/I.

Figure 4 The top end of PRTV is the ISBL interpreter. It has six main components, as shown in the diagram.



Incoming ISBL statements are analyzed by the ISBL syntax analyzer. Control statements such as LOGON, or the command to transfer a relation to or from the data base, are passed to the directory routines. Calls to user functions are passed to the user function control, a small interface routine which ensures that the required routine is properly linked and then translates parameters and passes control to the user function.

Incoming assignment statements require calls to the directory to resolve the names, then to the relational semantic routines to check and compile the relational operations, and finally to the directory to associate the result of the expression with the target.

The relational file control mostly translates ISBL-level calls directly to their CIL equivalents, with some checking and conversion. However, the command READ OPEN is presented with an ISBL expression that has to be evaluated by the rest of the compiler. Similarly, WRITE CLOSE has to call the directory routines to bind the brick to the required relation name.

The optimizer reorganizes cilstrings before they are submitted to the bottom end. It carries out transformations that allow complex streams to be more quickly evaluated. An example is rearranging a string so that selected operations are carried out as soon as possible.

The design of the top end is fairly straightforward. The only points of special interest are the use of complex mappings between relations and storage structures, and the function of the optimizer.

mapping from relations to storage structures

The values of relations stored in the directory are held in *relation control blocks*, each of which has two parts. One part is a cilstring which can be passed to the bottom end to realize a set of records. The other part, called a *domain list*, allows the top end to interpret the records as tuples. The domain list contains the degree of the relation, and, for each component, the domain from which it is drawn and its data type.

The use of control blocks allows for complex mapping between the user's (relational) view and the storage (brick and stream) view of data. It also simplifies the implementation of defined relations, which are mappings between different user views.

The main directory, the relation index, is a mapping from the user's name for a relation to a location containing the relation control block. The relational semantic routines act only on relation control blocks. They accept relation control blocks as parameters and use them to check the validity of the operation and produce a relation control block for the result. When an assignment is made, operations are carried out solely on the directories and relation control blocks, not on the bottom-end data. Operations on bottom-end data are only carried out in four cases: when the user lists the relation, when the user opens the relation as a relational file, when the user asks for the cardinality of a relation, and when the user explicitly requests that the relation be stored as a brick, using the command

KEEP <relation name>.

The delay in executing operations on bulk data is called *deferred operation.*

The cilstring used in a relation control block is an extension of the cilstring passed to the bottom end. It expresses the binding-by-name capabilities of ISBL by including the name in parentheses. Thus the statement A=N!B+N!C creates the cilstring +(B)(C) for A.

optimization

A set of stream operations often can be reorganized into another set that gives the same result but takes less time to execute. For example, usually it is quicker to make selections from two streams and join the result than to join the streams and select from the result. The optimization code reorganizes cilstrings accordingly, carrying out the reorganization on a tree form of the string.

Optimization includes the following activities:

- Filters are moved as far down the tree as possible, causing the selections to be executed as early as possible.

- Projections of projections are merged into one projection.

- Projections involving sorts are moved as far toward the top of the tree as possible, for latest possible execution.

- Projections not involving sorts are moved as far toward the bottom of the tree as possible.

- Expressions involving several set operators are reorganized according to such standard rules as commutativity and distribution. The sizes of the bricks are used to optimize this reorganization. Estimates are made of the sizes of the intermediate values. At present, no statistics are kept to help make these estimates more accurate.

- A search is made for common subtrees within the tree. The common value may be realized as a brick, preventing duplication of the operations. Often the cost of creating a brick is greater than the cost of repeating the operations, however, so the optimizer estimates both costs and chooses the cheaper alternative.

- The optimizer also chooses among alternative implementations of the relational operations. For example, the operation *join* can be implemented as either a collate or a double loop.[5, 6] The collate is quicker but requires suitably sorted input. If the input data are not sorted, the optimizer chooses whether to sort and merge them or to use the double-loop implementation. Another choice may be possible with *selection*, which sometimes can be implemented using indices held for bricks.

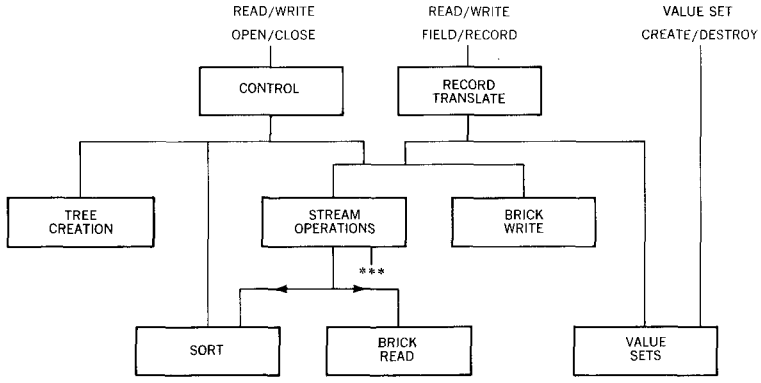
- In future, the optimizer will handle the use of secondary inversions for both selections[7] and joins.[5, 8]

An interesting optimization occurs when a selection is carried out on the result of a join. The selection is split four ways. One part is applied to the left-hand input of the join and one to the right. A third part is used by the join itself to drive collation, and the final part is applied after the join.

The optimizer operates effectively only because of the use of deferred operations. It allows cilstrings to become more complex than those caused by a user's entering a single input line. Thus, optimization occurs over an entire group of user statements while giving the impression of immediate execution of each statement. Deferred operation makes possible the use of optimizing compiler techniques in an interpretive environment.

Further details of optimization are given by Hall[9] and by Smith and Chang.[10]

Figure 5 The bottom end of PRTV contains subroutines that handle records representing tuples.



the bottom end

The bottom end of PRTV (see Figure 5) is implemented in a mixture of PL/I and System/370 assembler language. The following discussion covers the format of the major data structures—the brick and the value set—and then outlines their use in providing CIL interface functions. The discussion is intended not to give full details, but only to outline major points.

bricks

A brick is a stored sequential file. Techniques that make for the efficient storing and retrieving of bricks include the use of a standard format, the blocking of data, the sorting, suppression, and compression of data, and the use of page indices. Records are stored in a standard format throughout the bottom end to simplify access and manipulation routines. Blocking is intended to reduce the number of disk accesses. Sorting makes the execution of stream operations more efficient, and it allows for the suppression of duplicate leading fields. Data compression, applied on a field-by-field basis, recovers space lost in the sometimes apparently extravagant use of the eight-byte format. Page indices provide for faster access in the major sort field.

Partial inversions are being implemented in the form of binary bricks between the value in a given field and the identifiers of records containing that value.

Blocking is designed to save disk access time. The first record on each page is written in full at the head of the page. The page size is chosen when a data base is first formatted. The remaining records are suppressed and compressed and written sequentially onto the page until no more will fit. This procedure allows from one to more than 1000 records to be stored on a page.

Figure 6 An example of data compression as carried out in PRTV, showing how the value 257 is compressed against the value 515.

```
257     | 01000011 | 00010000 | 00010000 | 00000000 |···
|
515     | 01000011 | 00100000 | 00110000 | 00000000 |···
|
EOR     | 00000000 | 00110000 | 00100000 | 00000000 |···
|
RESULT  | 01100000 | 00110000 | 00100000 |
                |          |          |
              bit map    non-zero bytes
```

Permanently stored bricks are always sorted. The only bricks that are not sorted are those that have just been written by the CIL user. They are always sorted before being made permanent.

Data suppression is intended to eliminate CPU and I/O time and storage associated with certain redundant data. In each record, leading fields are suppressed if they are identical to the corresponding fields of the previous record. The remaining fields are preceded by a value denoting the number of fields explicitly held. For example, consider a file of degree 3:

1 1 1
1 1 2
1 2 2
2 1 2

In this case, only the following fields would be held:

  1 1 1
1   2
2   2 2
3 2 1 2

The underscored values above indicate the number of fields.

Compression is carried out as follows: The first record in each block is stored complete, but in subsequent records, fields that have not been suppressed are compressed against the corresponding fields of the first record. This is done by an exclusive OR of the values in the two records. If the values are similar, the result will contain several zero bytes. A bit map indicating the zero bytes is stored, followed by the nonzero bytes. An example is shown in Figure 6, where the value 257 is compressed against the value 515.

Compression is needed because of the eight-byte format. The technique used in PRTV was chosen to give good results with small integers and value-set indentifiers.

Page indices are designed to provide fast access for certain selection operations. The data pages of a brick are held in an

array, along with the value of the first field in the first and last records on each page.

value sets

Relations often have to hold character-string values. In PRTV these values are not held in the stream representation of the relations, but are stored on disk in value sets. The stream contains an identifier for the value. This system has the following advantages:

- records in a stream are of fixed length;
- long character strings are held only once in the system, so only their identifiers are duplicated if several bricks contain the same string.

On the other hand, the use of value sets has the following disadvantages:

- the manipulation of character strings is inefficient if they are used in only one brick;
- stream operations for such functions as substring tests are inefficient because they do not have convenient access to the values;
- it is difficult to present a properly sorted relational file unless the value-set identifiers are arranged to sort in the same order as the strings they represent;
- it becomes difficult to tell when a character string is no longer in use by any brick, and thus to purge it from the system.

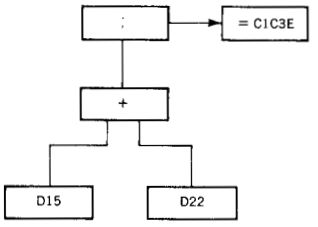
PRTV stores one value set for each domain of character data type created by the user, thereby establishing several small value sets instead of one large one. The advantage is speed of access from identifier to value and vice versa. Having several small value sets, however, makes retrospective merging of domains very expensive, and this operation is not supported in PRTV.

The current PRTV value-set scheme is based on unbalanced trees with implicit identifiers. This scheme is biased toward value-at-a-time retrieval of values from identifiers and does not maintain a helpful sort order of identifiers. The identifiers take up six bytes, an unnecessarily large amount of storage that is often reduced by compression. Short values are not indirectly coded but are held in the eight bytes with one byte reserved for the length.

trees, application, stream operations

When a complex stream is to be read, or turned into a brick, operations have to be carried out on the explicit records. These are called *stream operations*. Six of them correspond to the supported relational operations, one invokes tuple-at-a-time user extensions, one is a read operation, and one is a sort. Each works on a node of a tree using input from the subnodes. At the bottom of the tree are the nodes for the read operation, which simply reads records (usually sequentially) out of bricks.

Figure 7 Example of a tree, showing CIL ;+D15D22 =C1C3E.



To use the stream operations, a tree must be set up first, and then the operations invoked. The operations take data from the data base at the leaf nodes, and the data flow up the tree, the final result appearing at the top (see Figure 7).

The realization of the set of tuples at the top of a tree is the *explication* of the tree. As the tree is set up, certain operations are performed to make its explication more efficient. For example, filters are converted to forms that are easier to apply. Each node of the tree contains pointers to subnodes, filters, and so on. There is also space for the current record.

In most cases, stream operations take sorted input and develop sorted output, making it unnecessary to fully realize the intermediate streams that are passed from one node to another. Records then can be passed up the tree a few at a time, as requested by the upper node. PRTV does this whenever possible.

Sometimes intermediate streams do have to be fully realized and stored, as in a reorder, which does not necessarily produce sorted output. The output must be fully worked out and sorted before any records are passed up the tree. A node where an intermediate stream must be explicated is called a *break point* of a tree.

The code that carries out the conversion from CIL to tree is quite straightforward, as is that which carries out the stream operations. No details are included here.

basic operations of the bottom end

The simplest bottom-end operation is writing a brick. The OPEN command creates a control block which is used to identify the brick being written. As each record is entered, it is translated using value-set conversion for characters. This procedure puts it into the standard form for writing a brick. The brick is sorted, if necessary, when writing is closed.

Reading a stream and converting a stream to a brick require the building of a tree. In reading, the tree is built when the stream is opened. The records are explicated using the tree, then translated to user format before being passed out of the bottom end.

When a stream is converted to a brick, the explicated records simply are written to the brick without the need for value set translation.

PRTV background

PRTV is based on the concise, formal definition of relations proposed by Hall, Hitchcock, and Todd.[3] It is from that definition that the terms *selector* and *component*, as used in this paper, are taken. The definition proposed by Hall et al. solves problems of role names and domain name inheritance discussed by Codd[1] and clears up the confusion resulting from his use of *domain* for both underlying set and component of a relation.

Two other features of PRTV were discussed by Hall et al. The use of a relation as the graph of a procedure is the basis of the tuple-at-a-time extension; and the equijoin and generalized difference operations are based on operations described by Hall et al. These operations are more convenient than the operations used in most forms of the relational algebra.

The use of normal forms is discussed by Codd.[11] PRTV understands tuple elements only as atomic objects; it is possible for a general extension to use objects as relation names and thus to simulate a violation of first normal form. A system cannot check for third normal form, as that form is only an intensional property. It is recommended that users of PRTV keep their relations in third normal form in most cases.

optimization

The optimization carried out by PRTV was first suggested by M. G. Notley, but the details are the work of Hall.[9] Similar ideas have been proposed by Smith and Chang.[10]

Work of an entirely different nature on optimization of relational expressions has been done by Astrahan and Chamberlin,[7] who considered using inversions to optimize a smaller class of queries. It is hoped to include some of the results of their work in PRTV.

Current work at Peterlee is being directed toward optimization on the data-base scale rather than on the scale of a single query.[12] This optimization will not be entirely automatic; a data-base administrator interface is anticipated. Commands through the interface will affect only the efficiency of queries and programs, not the results obtained.

file structures

There are no particularly novel features in the file structures of PRTV. The implementation of bricks is much more conventional than the structures used in many "relational" access methods.[13]

The conventional indexing features are simpler than those of VSAM.[14] Value sets have been used in several previous systems, for example in a system described by Titman,[15] and as class relations in XRM.[13] The techniques used in data compression have been tailored to meet the needs of PRTV, but the basic concepts are from Titman's system. They have also been used in VSAM.

applications

ISBL is not designed as an end-user query language. PRTV is used either as a stand-alone system for users with a sufficient knowledge of algebra, or else as a data-base subsystem. No other relational system has emphasized the extensibility problems associated with subsystem use. (Many workers ignore the extensibility problem, forgetting that Codd's proof of completeness expressly covers only the limited *relational* completeness.[16])

PRTV has been used for several stand-alone applications at Peterlee, including a library system. The stand-alone version is used also for demonstration and research at several other IBM locations.

A major part of PRTV provides a data subsystem for the Urban Management System (UMS).[17] In an application of UMS,[18] a data base of 60 megabytes was created with relations of more than 60 000 tuples and degrees as high as 200. This data base was used regularly for over a year and is thought to be by far the largest relational data base to have had regular use.

Another system that incorporates PRTV as a subsystem is LEGOL,[19] used for computer-aided definition of statute laws. The data base holds a formal version of the rules of the laws as well as test cases.

A program has been written that translates a language similar to SEQUEL[20] into ISBL. The language has the same power as the SEQUEL query language but avoids the use of tuple variables. The translator comprises 40 lines of macroprocessor code.

Summary

high-level view of data

PRTV presents all data to the user in a standard form—that of relations. The relations are named in a flexible manner, and major operations on large volumes of data are carried out in a single statement. Therefore it is particularly easy to learn the system and write and debug applications.

multiple views of data

The bind-by-name facility of PRTV allows a user to take different views of the same data (or use various arrangements of it) for different applications, and it enables users to share data. Different users can have different data views, and entire areas of sensitive data can be withheld from unauthorized users.

extensibility

PRTV provides convenient exits so a user or system administrator can extend the function of the system. Extensions can make use of both relational views and more conventional sequential-file views of data. PRTV's extensibility enables the system to be used in a wide range of data-base applications.

a multilevel system

PRTV does not try to imitate the relational view of data directly at the storage level, and there is not a simple correspondence between relations and the stored files. Therefore the system can be implemented using efficient filing techniques such as sorting, compression, and indexing, without burdening the user with storage details.

optimization

Optimizing-compiler techniques aid in the efficient execution of apparently complex operations. Thus the user need not specify an especially efficient solution to his problem, but rather a convenient and readable one. Associated with the flexible file structures in PRTV is the use of trees to represent particular combinations of structures. The trees drive the explication of entire sets of tuples, so decisions on access paths can be made at a late stage, and optimization is based on the best data. At the same time, the use of trees reduces the number of decisions to be made as each record is processed.

ACKNOWLEDGMENTS

The basic design of PRTV was originated by Garth Notley. Many persons have been involved in coding the prototype, especially Alan Morrish, James Considine, and Patrick Hall. These and many others have assisted in the associated research. Terrence Rogers has managed the project throughout and has made many suggestions concerning the research, the prototype, and this description. The ideas received from the users of the system have been welcomed most of all.

CITED REFERENCES

1. E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM* **13,** No. 6, 377–387 (1970).
2. M. G. Notley, *The Peterlee IS/1 system,* IBM UK Scientific Centre Report 18, Peterlee, England (1972).
3. P. A. V. Hall, P. Hitchcock, and S. J. P. Todd, "An algebra of relations for machine computation," *Conference Record of the Second ACM Symposium on Principles of Programming Languages* (Palo Alto), 225–232 (1975).
4. S. Mandil, *The MP/3 Macroprocessor,* IBM UK Scientific Centre Report 44, Peterlee, England (1973).
5. S. J. P. Todd, *Implementation of the join operator in relational data bases,* IBM UK Scientific Centre Technical Note 15, Peterlee, England (1974).
6. L. R. Gotlieb, "Computing joins of relations," *Proceedings, ACM SIGMOD International Conference on Management of Data* (San Jose), 55–63 (1975).
7. M. M. Astrahan and D. D. Chamberlin, "Implementation of a structured English query language," *Communications of the ACM* **18,** No. 10, 580–588 (1975).

8. F. P. Palermo, "A data base search problem," *Information Systems, COINS IV* (Proceedings, Fourth International Symposium on Computer and Information Sciences, Miami Beach, 1972), 67–101, Plenum Press, New York (1974).
9. P. A. V. Hall, "Optimization of a single expression in a relational data base system," *IBM Journal of Research and Development* **20,** No. 3, 244–257 (1976).
10. J. M. Smith and P. Y. Chang, "Optimizing the performance of a relational algebra database interface," *Communications of the ACM* **18,** No. 10, 568–579 (1975).
11. E. F. Codd, "Further normalization of the data base relational model," *Courant Computer Science Symposium 6, Data Base Systems* (New York, 1971), 33–64, Prentice Hall, Englewood Cliffs, New Jersey (1972).
12. P. A. V. Hall and S. J. P. Todd, *Data base administrator facilities in PRTV*, IBM UK Scientific Centre Technical Note 22, Peterlee, England (1975).
13. R. A. Lorie, *XRM—an extended (n-ary) relational memory*, IBM Scientific Center Technical Report G320-2096, Cambridge, Massachusetts (1974).
14. R. E. Wagner, "Indexing design considerations," *IBM Systems Journal* **12,** No. 4, 351–367 (1973).
15. P. Titman, "An experimental data base system using binary relations," *Proceedings, IFIP Working Conference on Data Base Management* (Corsica), North Holland Publishing Co., Amsterdam, The Netherlands (1974).
16. E. F. Codd, "Relational completeness of data base sublanguages," *Courant Computer Science Symposium 6, Data Base Sytstems* (New York, 1971), 65–98, Prentice Hall, Englewood Cliffs, New Jersey (1972).
17. B. K. Aldred and B. S. Smedley, *An urban management system—general overview*, IBM UK Scientific Centre Report 53, Peterlee, England (1974).
18. B. S. Smedley, *An urban management system and geographic data processing in urban planning*, a joint project between the Greater London Council (Intelligence Unit) and IBM UK Scientific Centre, IBM UK Scientific Centre Report 79, Peterlee, England (to be published).
19. R. K. Stamper, *The LEGOL project, a survey*, IBM UK Scientific Centre Report 81, Peterlee, England (to be published).
20. D. D. Chamberlin and R. F. Boyce, "SEQUEL: a structured English query language," *Proceedings, ACM SIGMOD Workshop on Data Description, Access and Control* (Ann Arbor), 249–264 (1974).

Reprint Order No. G321-5038.